Oracle® TimesTen In-Memory Database

Java Developer's Guide

Release 11.2.1

E13068-08

March 2012

ORACLE®

Oracle TimesTen In-Memory Database Java Developer's Guide, Release 11.2.1

E13068-08

Contents

3 Using JMS/XLA for Event Management

4 Distributed Transaction Processing: JTA

5 Application Tuning

6 JMS/XLA Reference

Index

Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

TimesTen supports a variety of programming interfaces, including JDBC (Java Database Connectivity) and PL/SQL (Oracle procedural language extension for SQL).

This preface covers the following topics:

Audience

This guide is for anyone developing or supporting applications that use TimesTen through JDBC.

In addition to familiarity with JDBC, you should be familiar with TimesTen, SQL (Structured Query Language), and database operations.

Related documents

TimesTen documentation is available on the product distribution media and on the Oracle Technology Network:

`http://www.oracle.com/technetwork/database/timesten/documentation/`

Javadoc for standard JDBC classes and interfaces is available at the following location:

`http://download.oracle.com/javase/1.5.0/docs/api/`

Oracle documentation is also available on the Oracle Technology network. This may be especially useful for Oracle features that TimesTen supports but does not attempt to fully document:

`http://www.oracle.com/technetwork/database/enterprise-edition/documentation/`

In particular, the following Oracle document may be of interest.

- *Oracle Database SQL Language Reference*
- *Oracle Database JDBC Developer's Guide*

Conventions

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows applies to all supported Windows platforms. The term UNIX applies to all supported UNIX and Linux platforms. Refer to the "Platforms" section in *Oracle TimesTen In-Memory Database Release Notes* for specific platform versions supported by TimesTen.

> **Note:** In TimesTen documentation, the terms "data store" and "database" are equivalent. Both terms refer to the TimesTen database unless otherwise noted.

This document uses the following text conventions:

Convention	Meaning
italic	Italic type indicates terms defined in text, book titles, or emphasis.
`monospace`	Monospace type indicates code, commands, URLs, class names, interface names, method names, function names, attribute names, directory names, file names, text that appears on the screen, or text that you enter.
`italic monospace`	Italic monospace type indicates a placeholder or a variable in a code example for which you specify or use a particular value. For example: `Driver=`*`install_dir`*`/lib/libtten.sl` Replace *`install_dir`* with the path of your TimesTen installation directory.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line. An ellipsis in a code example indicates that what is shown is only a partial example.
%	The percent sign indicates the UNIX shell prompt.

TimesTen documentation uses these variables to identify path, file and user names:

Convention	Meaning
`install_dir`	The path that represents the directory where TimesTen is installed.
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique instance name. This name appears in the install path.
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
`release` or *`rr`*	Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.

Convention	Meaning
jdk_version	One or two digits that represent the version number of a major JDK release. For example, 14 is for JDK 1.4 and 5 is for JDK 5.
DSN	TimesTen data source name (for the TimesTen database).

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Technical support

For information about obtaining technical support for TimesTen products, go to the following Web address:

http://www.oracle.com/support/contact.html

What's New

This section summarizes the new features and functionality of Oracle TimesTen In-Memory Database Release 11.2.1 that are documented in this guide, providing links into the guide for more information.

New features in Release 11.2.1.7.0

- `CALL` for PL/SQL procedures and functions

  TimesTen now supports `CALL` syntax from any of its programming interfaces to call PL/SQL procedures and functions (in addition to `CALL` syntax to call TimesTen built-in procedures, which was already supported).

  See "Using CALL to execute procedures and functions" on page 2-24.

New features in Release 11.2.1.6.0

- User-specified parallel replication

  For applications that have very predictable transactional dependencies and do not require the commit order on the replica database to be the same as that on the originating database, TimesTen supports *parallel replication*. This feature allows replication of multiple user-specified *tracks* of transactions in parallel.

  See "Setting up user-specified parallel replication" on page 2-28.

New features in Release 11.2.1.4.0

- Support for Java 6

  See "Setting the environment for Java development" on page 1-1 regarding the class path.

- Additional rowid support (for use with Java 6)

  While rowids have been supported throughout the 11.2.1 release cycle as noted below, the 11.2.1.4.0 release adds support for the `java.sql.RowId` interface and `Types.ROWID` type.

  See "Working with rowids" on page 2-21.

- Synonyms

  TimesTen supports private and public synonyms (aliases) for database objects such as tables, views, sequences, and PL/SQL objects. See "Working with synonyms" on page 2-22.

New features in Release 11.2.1.1.0

- Quick Start demos

  This release includes an optional Quick Start feature with introductory information and some new or reworked demo applications. Note that the demos have mostly the same names as in earlier releases, but in a different location.

  See "About the TimesTen Java demos" on page 1-2 and *install_dir*/quickstart.html in your installation.

- Access control

  Perhaps the most significant overall change to previous functionality in this release is access control. TimesTen has new features to control database access with object-level resolution for database objects such as tables, views, materialized views, and sequences. This also affects access to certain TimesTen built-in procedures, utilities, and database and connection attributes.

  See "Considering TimesTen features for access control" on page 2-29. For general information, see "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide*.

- Output parameters

  TimesTen now supports `OUT` and `IN OUT` parameters for your database operations.

  See "Working with OUT and IN OUT parameters" on page 2-15.

- Duplicate parameters

  TimesTen now supports either of two modes for binding duplicate parameters in a SQL statement. Use the `DuplicateBindMode` general connection attribute to choose between Oracle mode and traditional TimesTen mode.

  See "Binding duplicate parameters in SQL statements" on page 2-17.

- REF CURSORs

  REF CURSOR is a PL/SQL concept, where a REF CURSOR is a handle to a cursor over a SQL result set and can be passed between PL/SQL and an application.

  See "Working with REF CURSORs" on page 2-18.

- Automatic client failover

  Automatic client failover, used in High Availability scenarios when failure of a TimesTen node results in failover (transfer) to an alternate node, automatically reconnects applications to the new node. TimesTen provides features that allow applications to be alerted when this happens, so they can take any appropriate action.

  See "JDBC support for automatic client failover" on page 2-33.

- DML returning (`RETURNING INTO` clause)

  You can use a `RETURNING INTO` clause, referred to as *DML returning*, with an `INSERT`, `UPDATE`, or `DELETE` statement to return specified items from a row that was affected by the action.

  See "Working with DML returning (RETURNING INTO clause)" on page 2-19.

- Rowids

Each row in a TimesTen database table has a unique identifier known as its rowid. TimesTen now supports Oracle-style rowids. An application can retrieve the rowid of a row from the `ROWID` pseudocolumn.

See "Working with rowids" on page 2-21.

- Execution time threshold for SQL statements

  You can configure TimesTen to write a warning to the support log and throw an SNMP trap when the execution of a SQL statement exceeds a specified time duration, in seconds. This feature was added in a 7.0.x maintenance release but not documented in this manual. Note that this feature is similar to but differs from the previously existing timeout value for SQL statements.

  See "Setting a timeout or threshold for executing SQL statements" on page 2-26.

- JMS/XLA replicated bookmarks

  If you are using an active standby pair replication scheme, you now have the option of using replicated bookmarks. For a replicated bookmark, operations on the bookmark are replicated to the standby database as appropriate. This allows more efficient recovery of your bookmark positions when failover occurs.

  See the section on replicated bookmarks under "XLA bookmarks" on page 3-4.

- Indication of XLA updates due to cascading deletes or aging

  TimesTen indicates if an XLA update was generated as part of a cascading delete or aging operation through new XLA flags.

  See "XLA flags" on page 6-2.

1

Java Development Environment

This chapter provides information about the Java development environment and related considerations. It includes the following topics:

- Installing TimesTen and the JDK
- Setting the environment for Java development
- Compiling Java applications
- About the TimesTen Java demos

Installing TimesTen and the JDK

Install and configure TimesTen for your environment, as described in *Oracle TimesTen In-Memory Database Installation Guide*, and the Java JDK, as described in your Java installation documentation. As you set up a Java development environment, the topics of particular interest in *Oracle TimesTen In-Memory Database Installation Guide* include the following:

- "Java environment variables"
- "Other Client/Server settings"
- "Environment variables"

After you have installed and configured TimesTen, create a database DSN as described in "Managing TimesTen Databases" in *Oracle TimesTen In-Memory Database Operations Guide*. The topics of particular interest include the following:

- "Connecting using the TimesTen JDBC driver and driver manager"
- "Overview of user and system DSNs"
- "Defining DSNs for direct or client/server connections"
- "Thread programming with TimesTen"
- "Creating a Data Manager DSN on UNIX" or "Creating a Data Manager DSN on Windows"

Setting the environment for Java development

Before you begin developing Java applications for TimesTen, you must set your environment appropriately. This includes the following considerations:

- Environment variables must be set appropriately. See "Java environment variables" in *Oracle TimesTen In-Memory Database Installation Guide* for more information

about environment variables for Java, including discussion of the `PATH`, `CLASSPATH`, `THREAD_FLAGS`, and shared library path environment variables.

- TimesTen includes Oracle Instant Client, which is required for certain JDBC features and operations.

Use the appropriate `ttenv` script to set up environment variables and runtime access to the Instant Client.

On UNIX platforms, execute one of the following scripts.

```
install_dir/bin/ttenv.sh
install_dir/bin/ttenv.csh
```

On Windows, run the following:

```
install_dir\bin\ttenv.bat
```

Compiling Java applications

"Java environment variables" in *Oracle TimesTen In-Memory Database Installation Guide* discusses the `CLASSPATH` setting for compiling Java applications in TimesTen.

Compiling any Java application requires the JAR file appropriate for your JDK to be in your classpath. In TimesTen, the following are for JDK 5.0 and JDK 6.0, respectively:

```
install_dir/lib/ttjdbc5.jar
install_dir/lib/ttjdbc6.jar
```

In addition, compiling any JMS/XLA application requires the following to be in your classpath:

```
install_dir/lib/timestenjmsxla.jar
install_dir/3rdparty/jms1.1/lib/jms.jar
install_dir/lib/orai18n.jar
```

About the TimesTen Java demos

After you have configured your Java environment, you can confirm that everything is set up correctly by compiling and running the TimesTen Quick Start demo applications. Refer to the Quick Start welcome page at *install_dir*`/quickstart.html`, especially the links under SAMPLE PROGRAMS, for information about the following:

- Demo schema and setup

  The `build_sampledb` script creates a sample database and demo schema. You must run this before you start using the demos.

- Demo environment and setup

  The `ttquickstartenv` script, a superset of the `ttenv` script generally used for TimesTen setup, sets up the demo environment. You must run this each time you enter a session where you want to compile and run any of the demos.

- Demos and setup

  TimesTen provides demos for JDBC and JMS/XLA under the `quickstart/sample_code` directory. For instructions on compiling and running the demos, see the README file or files in the subdirectories.

- What the demos do

A synopsis of each demo is provided when you click **JDBC (Java)** under SAMPLE PROGRAMS. The TimesTen basic Java demos are named `level1`, `level2`, `level3`, and `level4`. Data files for the `level` demos are in the `jdbc/datfiles` directory.

Note: All of the `level` demos support both direct and client/server connections to the database.

2

Working with TimesTen Databases

This chapter describes the basic procedures for writing a Java application to access data. Before attempting to write a TimesTen application, be sure you have completed the following prerequisite tasks:

Prerequisite task	What you do
Create a database.	Follow the procedures described in "Managing TimesTen Databases" in *Oracle TimesTen In-Memory Database Operations Guide*.
Configure the Java environment.	Follow the procedures described in "Setting the environment for Java development" on page 1-1.
Compile and execute the TimesTen Java demos.	Follow the procedures described in "About the TimesTen Java demos" on page 1-2.

After you have successfully executed the TimesTen Java demos, your development environment is set up correctly and ready for you to create applications that access a database.

Topics in this chapter are:

- Key JDBC classes and interfaces
- Managing TimesTen database connections
- Managing TimesTen data
- Using additional TimesTen data management features
- Considering TimesTen features for access control
- Handling errors
- JDBC support for automatic client failover

Key JDBC classes and interfaces

This section discusses important standard and TimesTen-specific JDBC packages, classes, and interfaces. The following topics are covered:

- Package imports
- Support for interfaces in the java.sql package
- Support for classes in the java.sql package
- Support for interfaces and classes in the javax.sql package

- TimesTen JDBC extensions
- Additional TimesTen classes and interfaces

For reference information on standard JDBC, see the following:

http://download.oracle.com/javase/1.5.0/docs/api/

For reference information on TimesTen JDBC extensions, refer to *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference*.

Package imports

You must import the standard JDBC package in any Java program that use JDBC:

```
import java.sql.*;
```

If you are going to use data sources or pooled connections, you must also import the standard extended JDBC package:

```
import javax.sql.*;
```

You must import the TimesTen JDBC package:

```
import com.timesten.jdbc.*;
```

To use XA data sources for JTA, you must also import the following TimesTen package:

```
import com.timesten.jdbc.xa.*;
```

Support for interfaces in the java.sql package

TimesTen supports the `java.sql` interfaces shown in Table 2–1.

Table 2–1 Supported java.sql interfaces

Interface in java.sql	Remarks on TimesTen support
`CallableStatement`	▪ You must pass parameters to `CallableStatement` by position, not by name. ▪ You cannot use SQL escape syntax. ▪ There is no support for `CLOB`, `BLOB`, `Array`, `Struct`, or `Ref`. ▪ There is no support for `Calendar` for `setDate()`, `getDate()`, `setTime()`, or `getTime()`.
`Connection`	▪ There is no support for savepoints.
`DatabaseMetaData`	▪ No restrictions.
`ParameterMetaData`	▪ The JDBC driver cannot determine whether a column is nullable and always returns `parameterNullableUnknown` from calls to `isNullable()`. ▪ The `getScale()` method returns 1 for `VARCHAR`, `NVARCHAR` and `VARBINARY` data types if they are `INLINE`. (Scale is of no significance to these data types.)
`PreparedStatement`	▪ There is no support for `getMetaData()` in `PreparedStatement`. ▪ There is no support for `CLOB`, `BLOB`, `Array`, `Struct`, or `Ref`. ▪ There is no support for `Calendar` for `setDate()`, `getDate()`, `setTime()`, or `getTime()`.

Table 2–1 (Cont.) Supported java.sql interfaces

Interface in java.sql	Remarks on TimesTen support
`Statement`	■ No restrictions. ■ See "Managing cache groups" on page 2-28 for special TimesTen functionality of the `getUpdateCount()` method with cache groups.
`ResultSet`	■ There *is* support for `getMetaData()` in `ResultSet`. ■ You cannot have multiple open `ResultSet` objects per statement. ■ You cannot specify the holdability of a result set, so a cursor cannot remain open after it has been committed. ■ There is no support for scrollable or updatable result sets. ■ There is no support for `CLOB`, `BLOB`, `Array`, `Struct`, or `Ref`. ■ There is no support for `Calendar` for `setDate()`, `getDate()`, `setTime()`, or `getTime()`. ■ See "Working with TimesTen result sets: hints and restrictions" on page 2-10.
`ResultSetMetaData`	■ The `getPrecision()` method returns 0 for undefined precision. ■ The `getScale()` method returns -127 for undefined scale.
`RowId`	**Note:** This support applies only when using Java 6 (`ttjdbc6.jar`). ■ The `ROWID` data type can be accessed using the `RowId` interface. ■ `OUT` and `IN OUT` rowids can be registered as `Types.ROWID`. ■ Metadata methods will return `Types.ROWID` and `RowId` as applicable.

Support for classes in the java.sql package

TimesTen supports the following `java.sql` classes:

- `Date`
- `DriverManager`
- `DriverPropertyInfo`
- `Time`
- `Timestamp`
- `Types`
- `DataTruncation`
- `SQLException`
- `SQLWarning`

Support for interfaces and classes in the javax.sql package

TimesTen supports the following `javax.sql` interfaces:

- `DataSource` is implemented by `TimesTenDataSource`.
- `PooledConnection` is implemented by `ObservableConnection`.
- `ConnectionPoolDataSource` is implemented by `ObservableConnectionDS`.

- `XADataSource` is implemented by `TimesTenXADataSource` (in package `com.timesten.jdbc.xa`).

  Important: The TimesTen JDBC driver itself does not implement a database connection pool. The `ObservableConnection` and `ObservableConnectionDS` classes simply implement standard Java EE interfaces, facilitating the creation and management of database connection pools according to the Java EE standard.

  A sample TimesTen connection pool package is shipped as part of the Quick Start demos. This is located in the following directory:

  install_dir`/quickstart/sample_code/jdbc/connectionpool`

TimesTen supports the following `javax.sql` event and listener:

- When using a `PooledConnection` instance, you can register a `ConnectionEventListener` instance to listen for `ConnectionEvent` occurrences.

  Note: It is permissible to register a `StatementEventListener` instance in TimesTen; however, `StatementEvent` instances are not supported.

TimesTen JDBC extensions

For most scenarios, you can use standard JDBC functionality as supported by TimesTen.

TimesTen also provides the following extensions in the `com.timesten.jdbc` package for TimesTen-specific features.

Table 2–2 TimesTen JDBC extensions

Interface	Extends	Remarks
`TimesTenConnection`	`Connection`	Provides capabilities such as prefetching rows to improve performance, listening to events for automatic client failover, and setting the track number for parallel replication. See "Fetching multiple rows of data" on page 2-10, "General Client Failover Features" on page 2-34, and "Setting up user-specified parallel replication" on page 2-28.
`TimesTenStatement`	`Statement`	Provides capabilities for specifying a query threshold. See "Setting a threshold value for SQL statements" on page 2-27.
`TimesTenPreparedStatement`	`PreparedStatement`	Supports DML returning. See "Working with DML returning (RETURNING INTO clause)" on page 2-19.

Table 2–2 (Cont.) TimesTen JDBC extensions

Interface	Extends	Remarks
`TimesTenCallableStatement`	`CallableStatement`	Supports PL/SQL REF CURSORs. See "Working with REF CURSORs" on page 2-18.

Additional TimesTen classes and interfaces

In addition to implementations discussed previously, TimesTen provides the following classes and interfaces in the `com.timesten.jdbc` package. Features supported by these classes and interfaces are discussed later in this chapter.

Additional TimesTen Interfaces

- Use `TimesTenTypes` for TimesTen type extensions (for REF CURSORs).
- Use `ClientFailoverEventListener` (and also the `ClientFailoverEvent` class below) for automatic client failover features. See "JDBC support for automatic client failover" on page 2-33.
- Use `TimesTenVendorCode` for vendor codes used in SQL exceptions.

Additional TimesTen Classes

- Use `ClientFailoverEvent` (and also the `ClientFailoverEventListener` interface above) for automatic client failover features.

Managing TimesTen database connections

The type of DSN you create depends on whether your application connects directly to the database or connects by a client. If you intend to connect directly to the database, create a DSN as described in "Creating a Data Manager DSN on UNIX" or "Creating a Data Manager DSN on Windows" in *Oracle TimesTen In-Memory Database Operations Guide*. If you intend to create a client connection to the database, create a DSN as described in "Creating and configuring Client DSNs on Windows" or "Creating and configuring Client DSNs on UNIX" in *Oracle TimesTen In-Memory Database Operations Guide*.

After you have created a DSN, the application can connect to the database. This section describes how to create a JDBC connection to a database using either the JDBC direct driver or the JDBC client driver.

The operations described in this section are based on the `level1` demo. Refer to "About the TimesTen Java demos" on page 1-2.

This following topics are covered here:

- Load the TimesTen driver
- Create a connection URL for the database and specify connection attributes
- Connect to the database
- Disconnect from the database
- Opening and closing a direct driver connection
- Access control for connections

Load the TimesTen driver

The TimesTen JDBC driver must be loaded before it is available for making database connections. The following is the TimesTen JDBC driver:

```
com.timesten.jdbc.TimesTenDriver
```

If you are using the `DriverManager` interface to connect to TimesTen, call the `Class.forName()` method to load the TimesTen JDBC driver. This method creates an instance of the TimesTen driver and registers it with the driver manager. If you are using the `TimesTenDataSource` interface, you are not required to call `Class.forName()`.

To identify and load the TimesTen driver:

```
Class.forName("com.timesten.jdbc.TimesTenDriver");
```

Note: If the TimesTen JDBC driver is not loaded, TimesTen returns an error when the application attempts to connect to the database.

Create a connection URL for the database and specify connection attributes

To create a JDBC connection, you must specify a TimesTen connection URL for the database. The format of a TimesTen connection URL is as follows:

```
jdbc:timesten:{direct|client}:dsn=DSNname;[DSNattributes;]
```

The default is `direct`.

For example, the following creates a direct connection to the sample database:

```
String URL = "jdbc:timesten:direct:dsn=sampledb_1121";
```

You can programmatically set or override the connection attributes in the DSN description by specifying attributes in the connection URL.

Refer to "Connection attributes for Data Manager DSNs or Server DSNs" in *Oracle TimesTen In-Memory Database Operations Guide* for introductory information about connection attributes. General connection attributes require no special privilege. First connection attributes are set when the database is first loaded, and persist for all connections. Only the instance administrator can load a database with changes to first connection attribute settings. Refer to "Connection Attributes" in *Oracle TimesTen In-Memory Database Reference* for specific information about any particular connection attribute, including required privilege.

For example, to set the `LockLevel` general connection attribute to 1, create a URL as follows:

```
String URL = "jdbc:timesten:direct:dsn=sampledb_1121;LockLevel=1";
```

Connect to the database

After you have defined a URL, you can use the `getConnection()` method of either `DriverManager` or `TimesTenDataSource` to connect to the database.

If you use the `DriverManager.getConnection()` method, specify the driver URL to connect to the database.

```
import java.sql.*;
...
Connection conn = DriverManager.getConnection(URL);
```

To use the TimesTenDataSource method getConnection(), first create a data source. Then use the TimesTenDataSource method setUrl() to set the URL and getConnection() to connect:

```
import com.timesten.jdbc.TimesTenDataSource;
import java.sql.*;
...
TimesTenDataSource ds = new TimesTenDataSource();
ds.setUrl("jdbc:timesten:direct:<dsn>");
Connection conn = ds.getConnection();
```

The TimesTen user name and password can be set in the DSN within the URL in the setUrl() call, but there are also TimesTenDataSource methods to set them separately, as well as to set the Oracle password (as applicable):

```
TimesTenDataSource ds = new TimesTenDataSource();
ds.setUser(myttusername);                    // User name to log in to TimesTen.
ds.setPassword(myttpwd);                     // Password to log in to TimesTen.
ds.setUrl("jdbc:timesten:direct:<dsn>");
ds.setOraclePassword(myorapwd);              // Password to log in to Oracle.
Connection conn = ds.getConnection();
```

Either the DriverManager.getConnection() method or the ds.getConnection() method returns a Connection object (conn in this example) that you can use as a handle to the database. See the level1 demo for an example on how to use the DriverManager method getConnection(), and the level2 and level3 demos for examples of using the TimesTenDataSource method getConnection(). Refer to "About the TimesTen Java demos" on page 1-2.

> **Note:** If the TimesTen JDBC driver is not loaded, TimesTen returns an error when the application attempts to connect to the database. See "Load the TimesTen driver" on page 2-6.

Disconnect from the database

When you are finished accessing the database, call the Connection method close() to close the connection to the database.

If an error has occurred, you may want to roll back the transaction before disconnecting from the database. See "Handling non-fatal errors" on page 2-30 and "Rolling back failed transactions" on page 2-33 for more information.

Opening and closing a direct driver connection

Example 2–1 shows the general framework for an application that uses the DriverManager class to create a direct driver connection to the sample database, execute some SQL, and then close the connection. See the level1 demo for a working example. (See "About the TimesTen Java demos" on page 1-2 regarding the demos.)

Example 2–1 Connecting, executing SQL, and disconnecting

```
String URL = "jdbc:timesten:dsn=sampledb_1121";
Connection conn = null;

try {
     Class.forName("com.timesten.jdbc.TimesTenDriver");
} catch (ClassNotFoundException ex) {
```

```
      // See "Handling errors" on page 2-30
}

try {
    // Open a connection to TimesTen
    conn = DriverManager.getConnection(URL);

    // Report any SQLWarnings on the connection
    // See "Reporting errors and warnings" on page 2-31

    // Do SQL operations
    // See "Managing TimesTen data" below

// Close the connection to TimesTen
   conn.close();

// Handle any errors
} catch (SQLException ex) {
    // See "Handling errors" on page 2-30
}
```

Access control for connections

Privilege to connect to a database must be explicitly granted to every user other than the instance administrator, through the `CREATE SESSION` privilege. This is a system privilege so must be granted by an administrator to the user, either directly or through the `PUBLIC` role. Refer to "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide* for additional information and examples.

Managing TimesTen data

This section provides detailed information on working with data in a TimesTen database. It includes the following topics:

- Executing simple SQL statements
- Working with TimesTen result sets: hints and restrictions
- Fetching multiple rows of data
- Binding parameters and executing statements
- Working with REF CURSORs
- Working with DML returning (RETURNING INTO clause)
- Working with rowids
- Committing or rolling back changes to the database
- Managing multiple threads
- Java escape syntax and SQL functions

Executing simple SQL statements

"Working with Data in a TimesTen Database" in *Oracle TimesTen In-Memory Database Operations Guide* describes how to use SQL to manage data. This section describes how to use the `createStatement()` method of a `Connection` instance, and the `executeUpdate()` or `executeQuery()` method of a `Statement` instance, to execute a SQL statement within a Java application.

Unless statements are prepared in advance, use the execution methods of a `Statement` object, such as `execute()`, `executeUpdate()` or `executeQuery()`, depending on the nature of the SQL statement and any returned result set.

For SQL statements that are prepared in advance, use the same execution methods of a `PreparedStatement` object.

The `execute()` method returns `true` if there is a result set (for example, on a `SELECT`) or `false` if there is no result set (for example, on an `INSERT`, `UPDATE`, or `DELETE`). The `executeUpdate()` method returns the number of rows affected. For example, when executing an `INSERT` statement, the `executeUpdate()` method returns the number of rows inserted. The `executeQuery()` method returns a result set, so it should only be called when a result set is expected (for example, when executing a `SELECT` statement).

Notes:

- See "Working with TimesTen result sets: hints and restrictions" on page 2-10 for details about what you should know when working with result sets generated by TimesTen.
- Access control privileges are checked both when SQL is prepared and when it is executed in the database. Refer to "Considering TimesTen features for access control" on page 2-29 for related information.

Example 2–2 Executing an update

This example uses the `executeUpdate()` method on the `Statement` object to execute an `INSERT` statement to insert data into the `customer` table in the current schema. The connection must also be opened, which is not shown.

```
Connection conn;
Statement stmt;
...
// [Code to open connection. See "Connect to the database" on page 2-6...]
...
try {
    stmt = conn.createStatement();
    int numRows = stmt.executeUpdate("insert into customer values"
                  + "(40, 'West', 'Big Dish', '123 Signal St.')");
}
catch (SQLException ex) {
    ...
}
```

Example 2–3 Executing a query

This example uses an `executeQuery()` call on the `Statement` object to execute a `SELECT` statement on the `customer` table in the current schema and display the returned `java.sql.ResultSet` instance:

```
Statement stmt;
. . . . . .
try {
  ResultSet rs = stmt.executeQuery("select cust_num, region, " +
                       "name, address from customer");
  System.out.println("Fetching result set...");
  while (rs.next()) {
    System.out.println("\n Customer number: " + rs.getInt(1));
```

```
    System.out.println(" Region: " + rs.getString(2));
    System.out.println(" Name: " + rs.getString(3));
    System.out.println(" Address: " + rs.getString(4));
    }
  }
catch (SQLException ex) {
  ex.printStackTrace();
}
```

Working with TimesTen result sets: hints and restrictions

Use `ResultSet` objects to process query results. In addition, some methods and built-in procedures return TimesTen data in the form of a `ResultSet` object. This section describes what you should know when using `ResultSet` objects from TimesTen.

- TimesTen does not support multiple open `ResultSet` objects per statement. TimesTen cannot return multiple `ResultSet` objects from a single `Statement` object without first closing the current result set.
- TimesTen does not support holdable cursors. You cannot specify the holdability of a result set, essentially whether a cursor can remain open after it has been committed.
- `ResultSet` objects are not scrollable or updatable, so you cannot specify `ResultSet.TYPE_SCROLL_SENSITIVE` or `ResultSet.CONCUR_UPDATABLE`.
- Use the `ResultSet` method `close()` to close a result set as soon as you are done with it. For performance reasons, this is especially important for result sets used for both read and update operations and for result sets used in pooled connections.
- Calling the `ResultSet` method `getString()` is more costly in terms of performance if the underlying data type is not a string. Because Java strings are immutable, `getString()` must allocate space for a new string each time it is called. Do not use `getString()` to retrieve primitive numeric types, like `byte` or `int`, unless it is absolutely necessary. For example, it is much faster to call `getInt()` on an integer column. Also see "Use the ResultSet method getString() sparingly" on page 5-3.

  In addition, it is generally true for dates and timestamps that `ResultSet` native methods `getDate()` and `getTimestamp()` will have better performance than `getString()`.
- Application performance is influenced by the choice of `get`*`XXX`*`()` calls and by any required data transformations after invocation.
- JDBC ignores the setting for the `ConnectionCharacterSet` attribute. It returns data in UTF-16 encoding.

Fetching multiple rows of data

Fetching multiple rows of data can increase the performance of an application that connects to a database set with Read Committed isolation level.

You can specify the number of rows to be prefetched as follows.

- Call the `Statement` or `ResultSet` method `setFetchSize()`. These are the standard JDBC calls, but the limitation is that they only affect one statement at a time.

- Call the `TimesTenConnection` method `setTtPrefetchCount()`. This enables a TimesTen extension that establishes prefetch on a connection level so that all of the statements on the connection use the same prefetch setting.

This section describes the connection-level prefetch implemented in TimesTen.

> **Note:** You can use the TimesTen prefetch count extension only with direct-linked applications.

When the prefetch count is set to 0, TimesTen uses a default value, depending on the isolation level you have set for the database. With Read Committed isolation level, the default prefetch value is 5. With Serializable isolation level, the default is 128. The default prefetch value is the optimum setting for most applications. Generally, a higher value may result in better performance for larger result sets, at the expense of slightly higher resource use.

To disable prefetch, set the prefetch count to 1.

Call the `TimesTenConnection` method `getTtPrefetchCount()` to check the current prefetch value.

Example 2–4 Setting a prefetch count

The following code uses a `setTtPrefetchCount()` call to set the prefetch count to 10, then uses a `getTtPrefetchCount()` call to return the prefetch count in the count variable.

```
TimesTenConnection conn =
    (TimesTenConnection) DriverManager.getConnection(url);

// set prefech count to 10 for this connection
conn.setTtPrefetchCount(10);

// Return the prefetch count to the 'count' variable.
int count = conn.getTtPrefetchCount();
```

Binding parameters and executing statements

This sections discusses how to bind input or output parameters for SQL statements. The following topics are covered:

- Preparing SQL statements and setting input parameters
- Working with OUT and IN OUT parameters
- Binding duplicate parameters in SQL statements
- Binding duplicate parameters in PL/SQL

> **Note:** Array binding, the ability to bind associative arrays (index-by tables) and varrays (variable size arrays) into PL/SQL statements, is not supported in TimesTen JDBC.

Preparing SQL statements and setting input parameters

SQL statements that are to be executed more than once should be prepared in advance by calling the `Connection` method `prepareStatement()`. For maximum performance, prepare parameterized statements.

Notes:

- It is generally true for time, dates, and timestamps that `PreparedStatement` native methods `setTime()`, `setDate()` and `setTimestamp()` will have better performance than `setString()`.
- Application performance is influenced by the choice of `set`*`XXX`*`()` calls and by any required data transformations before invocation.
- Access control privileges are checked both when SQL is prepared and when it is executed in the database. Refer to "Considering TimesTen features for access control" on page 2-29 for related information.
- For `TT_TINYINT` columns, use `setShort()` or `setInt()` instead of `setByte()` to realize the full range of `TT_TINYINT` (0-255).

Example 2–5 Prepared statement for querying

This example shows the basics of an `executeQuery()` call on a `PreparedStatement` object. It executes a prepared `SELECT` statement and displays the returned result set.

```
PreparedStatement pSel = conn.prepareStatement("select cust_num, " +
                         "region, name, address " +
                         "from customer" +
                         "where region = ?");
pSel.setInt(1,1);

try {
  ResultSet rs = pSel.executeQuery();

  while (rs.next()) {
    System.out.println("\n Customer number: " + rs.getInt(1));
    System.out.println(" Region: " + rs.getString(2));
    System.out.println(" Name: " + rs.getString(3));
    System.out.println(" Address: " + rs.getString(4));
  }
}
catch (SQLException ex) {
   ex.printStackTrace();
}
```

Example 2–6 Prepared statement for updating

This example shows how a single parameterized statement can be substituted for four separate statements.

Rather than execute a similar `INSERT` statement with different values:

```
Statement.execute("insert into t1 values (1, 2)");
Statement.execute("insert into t1 values (3, 4)");
Statement.execute("insert into t1 values (5, 6)");
Statement.execute("insert into t1 values (7, 8)");
```

It is much more efficient to prepare a single parameterized `INSERT` statement and use `PreparedStatement` methods `set`*`XXX`*`()` to set the row values before each execute.

```
PreparedStatement pIns = conn.PreparedStatement("insert into t1 values (?,?)");
```

```
pIns.setInt(1, 1);
pIns.setInt(2, 2);
pIns.executeUpdate();

pIns.setInt(1, 3);
pIns.setInt(2, 4);
pIns.executeUpdate();

pIns.setInt(1, 5);
pIns.setInt(2, 6);
pIns.executeUpdate();

pIns.setInt(1, 7);
pIns.setInt(2, 8);
pIns.executeUpdate();

conn.commit();
pIns.close();
```

TimesTen shares prepared statements automatically after they have been committed. For example, if two or more separate connections to the database each prepare the same statement, then the second, third, ... , *n*th prepared statements return very quickly because TimesTen remembers the first prepared statement.

Example 2–7 Prepared statements for updating and querying

This example prepares `INSERT` and `SELECT` statements, executes the `INSERT` twice, executes the `SELECT`, and prints the returned result set. For a working example, see the `level1` demo. (Refer to "About the TimesTen Java demos" on page 1-2 regarding the demos.)

```
Connection conn;
...
// [Code to open connection. See "Connect to the database" on page 2-6...]
...

// Disable auto-commit
conn.setAutoCommit(false);

    // Report any SQLWarnings on the connection
    // See "Reporting errors and warnings" on page 2-31

// Prepare a parameterized INSERT and a SELECT Statement
PreparedStatement pIns =
                  conn.prepareStatement("insert into customer values (?,?,?,?)");

PreparedStatement pSel = conn.prepareStatement
    ("select cust_num, region, name, " +
    "address from customer");

// Data for first INSERT statement
pIns.setInt(1, 100);
pIns.setString(2, "N");
pIns.setString(3, "Fiberifics");
pIns.setString(4, "123 any street");

// Execute the INSERT statement
pIns.executeUpdate();
```

```
// Data for second INSERT statement
pIns.setInt(1, 101);
pIns.setString(2, "N");
pIns.setString(3, "Natural Foods Co.");
pIns.setString(4, "5150 Johnson Rd");

// Execute the INSERT statement
pIns.executeUpdate();

// Commit the inserts
conn.commit();

// Done with INSERTs, so close the prepared statement
pIns.close();

// Report any SQLWarnings on the connection.
reportSQLWarnings(conn.getWarnings());

// Execute the prepared SELECT statement
ResultSet rs = pSel.executeQuery();

System.out.println("Fetching result set...");
while (rs.next()) {
    System.out.println("\n Customer number: " + rs.getInt(1));
    System.out.println(" Region: " + rs.getString(2));
    System.out.println(" Name: " + rs.getString(3));
    System.out.println(" Address: " + rs.getString(4));
}

// Close the result set.
rs.close();

// Commit the select - yes selects must be committed too
conn.commit();

// Close the select statement - we're done with it
pSel.close();
```

Example 2–8 Prepared statements for multiple connections

This example, prepares three identical parameterized `INSERT` statements for three separate connections. The first prepared `INSERT` for connection `conn1` is shared with the `conn2` and `conn3` connections and speeds up the prepare operations for `pIns2` and `pIns3`:

```
Connection conn1;
Connection conn2;
Connection conn3;
.....
PreparedStatement pIns1 = conn1.prepareStatement
                  ("insert into t1 values (?,?)");

PreparedStatement pIns2 = conn2.prepareStatement
                  ("insert into t1 values (?,?)");

PreparedStatement pIns3 = conn3.prepareStatement
                  ("insert into t1 values (?,?)");
```

Note: All tuning options, such as join ordering, indexes and locks, must match for the statement to be shared. Also, if the prepared statement references a temp table, it is only shared within a single connection.

Working with OUT and IN OUT parameters

"Preparing SQL statements and setting input parameters" on page 2-11 shows how to prepare a statement and set input parameters using `PreparedStatement` methods. TimesTen also supports `OUT` and `IN OUT` parameters, for which you use `java.sql.CallableStatement` instead of `PreparedStatement`, as follows.

1. Use the method `registerOutParameter()` to register an `OUT` or `IN OUT` parameter, specifying the parameter position (position in the statement) and data type.

   This is the standard method as specified in the `CallableStatement` interface:

   ```
   void registerOutParameter(int parameterIndex, int sqlType, int scale)
   ```

   Be aware, however, that if you use this standard version for `CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`, `BINARY`, or `VARBINARY` data, TimesTen will allocate memory to hold the largest possible value. In many cases this is wasteful.

   Instead, you can use the TimesTen extended interface `TimesTenCallableStatement`, which includes a `registerOutParameter()` signature that enables you to specify the maximum data length. For `CHAR`, `VARCHAR`, `NCHAR`, and `NVARCHAR`, the unit of length is number of characters. For `BINARY` and `VARBINARY`, it is bytes.

   ```
   void registerOutParameter(int paramIndex,
                             int sqlType,
                             int ignore, //This parameter is ignored by TimesTen.
                             int maxLength)
   ```

2. Use the appropriate `CallableStatement` method `set`*`XXX`*`()`, where *`XXX`* indicates the data type, to set the input value of an `IN OUT` parameter. Specify the parameter position and data value.

3. Use the appropriate `CallableStatement` method `get`*`XXX`*`()` to get the output value of an `OUT` or `IN OUT` parameter, specifying the parameter position.

Important: Check for SQL warnings before processing output parameters. In the event of a warning, output parameters are undefined. See "Handling errors" on page 2-30 for general information about errors and warnings.

Notes: In TimesTen:

- You cannot pass parameters to a `CallableStatement` object by name. You must set parameters by position. You cannot use the SQL escape syntax.
- The `registerOutParameter()` signatures specifying the parameter by name are not supported. You must specify the parameter by position.
- SQL structured types are not supported.

Example 2–9 Using an OUT parameter in a callable statement

This example shows how to use a callable statement with an `OUT` parameter. In the `TimesTenCallableStatement` instance, a PL/SQL block calls a function `RAISESAL` that calculates a new salary and returns it as an integer. Assume a `Connection` instance `conn`. (Refer to *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for information about PL/SQL.)

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.Types;
import com.timesten.jdbc.TimesTenCallableStatement;
...
// Prepare to call a PL/SQL stored procedure RAISESAL (raise salary)
CallableStatement cstmt = conn.prepareCall
                          ("BEGIN :newSalary := RAISESAL(:name, :inc); end;");

// Declare that the first param (newSalary) is a return (output) value of type int
cstmt.registerOutParameter(1, Types.INTEGER);

// Raise Leslie's salary by $2000 (she wanted $3000 but we held firm)
cstmt.setString(2, "LESLIE"); // name argument (type String) is the second param
cstmt.setInt(3, 2000); // raise argument (type int) is the third param

// Do the raise
cstmt.execute();

// Check warnings. If there are warnings, values of OUT parameters are undefined.
SQLWarning wn;
boolean warningFlag = false;
if ((wn = cstmt.getWarnings() ) != null) {
   do {
        warningFlag = true;
        System.out.println(wn);
        wn = wn.getNextWarning();
   } while(wn != null);
}

// Get the new salary back
if (!warningFlag) {
   int new_salary = cstmt.getInt(1);
   System.out.println("The new salary is: " + new_salary);
}

// Close the statement and connection
cstmt.close();
conn.close();
...
```

Binding duplicate parameters in SQL statements

TimesTen supports either of two modes for binding duplicate parameters in a SQL statement:

- Oracle mode, where multiple occurrences of the same parameter name are considered to be distinct parameters.
- Traditional TimesTen mode, as in earlier releases, where multiple occurrences of the same parameter name are considered to be multiple occurrences of the same parameter.

You can choose the desired mode through the `DuplicateBindMode` general connection attribute. `DuplicateBindMode=0` (the default) is for the Oracle mode, and `DuplicateBindMode=1` is for the TimesTen mode. Because this is a general connection attribute, different concurrent connections to the same database can use different values. Refer to "DuplicateBindMode" in *Oracle TimesTen In-Memory Database Reference* for additional information about this attribute.

The rest of this section provides details for each mode, considering the following query:

```
SELECT * FROM employees
  WHERE employee_id < :a AND manager_id > :a AND salary < :b;
```

> **Note:** This discussion applies only to SQL statements issued directly from JDBC (not through PL/SQL, for example).

Oracle mode for duplicate parameters In the Oracle mode, multiple occurrences of the same parameter name in a SQL statement are considered to be different parameters. When parameter position numbers are assigned, a number is given to each parameter occurrence without regard to name duplication. The application must, at a minimum, bind a value for the first occurrence of each parameter name. For any subsequent occurrence of a given parameter name, the application has the following choices.

- It can bind a different value for the occurrence.
- It can leave the parameter occurrence unbound, in which case it takes the same value as the first occurrence.

In either case, each occurrence still has a distinct parameter position number.

To use a different value for the second occurrence of `a` in the SQL statement above:

```
pstmt.setXXX(1, ...); /* first occurrence of :a */
pstmt.setXXX(2, ...); /* second occurrence of :a */
pstmt.setXXX(3, ...); /* occurrence of :b */
```

To use the same value for both occurrences of `a`:

```
pstmt.setXXX(1, ...); /* both occurrences of :a */
pstmt.setXXX(3, ...); /* occurrence of :b */
```

Parameter `b` is considered to be in position 3 regardless.

TimesTen mode for duplicate parameters In the TimesTen mode, SQL statements containing duplicate parameters are parsed such that only distinct parameter names are considered as separate parameters. The application binds a value only for each unique parameter, and no unique parameter can be left unbound.

Binding is based on the position of the first occurrence of a parameter name. Subsequent occurrences of the parameter name are bound to the same value, and are not given parameter position numbers.

For the SQL statement above, the two occurrences of `a` are considered to be a single parameter, so cannot be bound separately:

```
pstmt.setXXX(1, ...); /* both occurrences of :a */
pstmt.setXXX(2, ...); /* occurrence of :b */
```

Note that in the TimesTen mode, parameter `b` is considered to be in position 2, not position 3.

Binding duplicate parameters in PL/SQL

The preceding discussion does not apply within PL/SQL. Instead, PL/SQL semantics apply, whereby you bind a value for each unique parameter. An application executing the following block, for example, would bind only one parameter, corresponding to `:a`.

```
DECLARE
   x NUMBER;
   y NUMBER;
BEGIN
   x:=:a;
   y:=:a;
END;
```

An application executing the following block would also bind only one parameter:

```
BEGIN
   INSERT INTO tab1 VALUES(:a, :a);
END
```

And the same for the following `CALL` statement:

```
...CALL proc(:a, :a)...
```

An application executing the following block would bind two parameters, with `:a` as parameter #1 and `:b` as parameter #2. The second parameter in each `INSERT` statement would take the same value as the first parameter in the first `INSERT` statement, as follows.

```
BEGIN
   INSERT INTO tab1 VALUES(:a, :a);
   INSERT INTO tab1 VALUES(:b, :a);
END
```

Working with REF CURSORs

REF CURSOR is a PL/SQL concept, where a REF CURSOR is a handle to a cursor over a SQL result set and can be passed between PL/SQL and an application. In TimesTen, the cursor can be opened in PL/SQL, then the REF CURSOR can be passed to the application for processing of the result set.

An application can receive a REF CURSOR, as an `OUT` parameter, as follows:

1. Register the REF CURSOR `OUT` parameter as type `TimesTenTypes.CURSOR` (a TimesTen type extension), also specifying the parameter position of the REF CURSOR (position in the statement).

2. Retrieve the REF CURSOR using the `getCursor()` method defined by the `TimesTenCallableStatement` interface (a TimesTen JDBC extension), specifying the parameter position of the REF CURSOR. The `getCursor()` method is used like other `get`*`XXX`*`()` methods and returns a `ResultSet` instance.

   Important: For passing REF CURSORs between PL/SQL and an application, TimesTen supports only `OUT` REF CURSORs, from PL/SQL to the application, and supports a statement returning only a single REF CURSOR.

The following example demonstrates this usage.

Example 2–10 Using a REF CURSOR

This example shows how to use a callable statement with a REF CURSOR. In the `CallableStatement` instance, a PL/SQL block opens a cursor and executes a query. The `TimesTenCallableStatement` method `getCursor()` is used to return the cursor, which is registered as `TimesTenTypes.CURSOR`.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.ResultSet;
import com.timesten.jdbc.TimesTenCallableStatement;
import com.timesten.jdbc.TimesTenTypes;
...
Connection conn;
CallableStatement cstmt;
ResultSet cursor;
...
// Use a PL/SQL block to open the cursor.
cstmt = conn.prepareCall
             (" begin open :x for select tblname,tblowner from tables; end;");
cstmt.registerOutParameter(1, TimesTenTypes.CURSOR);
cstmt.execute();
cursor = ((TimesTenCallableStatement)cstmt).getCursor(1);

// Use the cursor as you would any other ResultSet object.
while(cursor.next()){
  System.out.println(cursor.getString(1));
}

// Close the statement
cstmt.close();
conn.close();
...
```

Working with DML returning (RETURNING INTO clause)

You can use a RETURNING INTO clause, referred to as *DML returning*, with an `INSERT`, `UPDATE`, or `DELETE` statement to return specified items from a row that was affected by the action. This eliminates the need for a subsequent `SELECT` statement and separate round trip, in case, for example, you want to confirm what was affected by the action.

With TimesTen, DML returning is limited to returning items from a single-row operation. The clause returns the items into a list of `OUT` parameters.

`TimesTenPreparedStatement`, an extension of the standard `PreparedStatement` interface, supports DML returning. Use the `TimesTenPreparedStatement` method `registerReturnParameter()` to register the return parameters.

```
void registerReturnParameter(int paramIndex, int sqlType)
```

As with the `registerOutParameter()` method discussed in "Working with OUT and IN OUT parameters" on page 2-15, this method includes a signature that enables you to optionally specify a maximum size for `CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`, `BINARY`, or `VARBINARY` data. This avoids possible inefficiency where TimesTen would otherwise allocate memory to hold the largest possible value. For `CHAR`, `VARCHAR`, `NCHAR`, and `NVARCHAR`, the unit of size is number of characters. For `BINARY` and `VARBINARY`, it is bytes.

```
void registerReturnParameter(int paramIndex, int sqlType, int maxSize)
```

Use the `TimesTenPreparedStatement` method `getReturnResultSet()` to retrieve the return parameters, returning a `ResultSet` instance.

Be aware of the following restrictions.

- The `getReturnResultSet()` method must not be invoked more than once. Otherwise, the behavior is indeterminate.
- `ResultSetMetaData` is not supported for the result set returned by `getReturnResultSet()`.
- Streaming methods such as `getCharacterStream()` are not supported for the result set returned by `getReturnResultSet()`.
- There is no batch support for DML returning.

SQL syntax and restrictions for the `RETURNING INTO` clause in TimesTen are documented as part of the "INSERT", "UPDATE", and "DELETE" documentation in *Oracle TimesTen In-Memory Database SQL Reference*.

Refer to "RETURNING INTO Clause" in *Oracle Database PL/SQL Language Reference* for general information about DML returning.

Important: Check for SQL warnings after executing the TimesTen prepared statement. In the event of a warning, output parameters are undefined. See "Handling errors" on page 2-30 for general information about errors and warnings.

Example 2–11 DML returning

This example shows how to use DML returning with a `TimesTenPreparedStatement` instance, returning the name and age for a row that is inserted.

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Types;
import com.timesten.jdbc.TimesTenPreparedStatement;

Connection conn;

...
```

```
// Insert into a table and return results
TimesTenPreparedStatement pstmt =
  (TimesTenPreparedStatement)conn.prepareStatement
  ("insert into tab1 values(?,?) returning name, age into ?,?");

// Populate table
pstmt.setString(1,"John Doe");
pstmt.setInt(2, 65);

/** register returned parameter
 * in this case the maximum size of name is 100 chars
 */
pstmt.registerReturnParameter(3, Types.VARCHAR, 100);
pstmt.registerReturnParameter(4, Types.INTEGER);

// process the DML returning statement
int count = pstmt.executeUpdate();

/* Check warnings; if there are warnings, values of DML RETURNING INTO
   parameters are undefined. */
SQLWarning wn;
boolean warningFlag = false;
if ((wn = pstmt.getWarnings() ) != null) {
  do {
    warningFlag = true;
    System.out.println(wn);
    wn = wn.getNextWarning();
  } while(wn != null);
}

if (!warningFlag) {
  if (count>0)
  {
    ResultSet rset = pstmt.getReturnResultSet(); //rset not null, not empty
    while(rset.next())
    {
      String name = rset.getString(1);
      int age = rset.getInt(2);
      System.out.println("Name " + name + " age " + age);
    }
  }
}
```

Working with rowids

Each row in a table has a unique identifier known as its *rowid*. An application can retrieve the rowid of a row from the `ROWID` pseudocolumn. A rowid value can be represented in either binary or character format, with the binary format taking 12 bytes and the character format 18 bytes.

For Java 6, TimesTen supports the `java.sql.RowId` interface and `Types.ROWID` type.

You can use any of the following `ResultSet` methods to retrieve a rowid:

- `byte[] getBytes(int` *`columnIndex`*`)`
- `String getString(int` *`columnIndex`*`)`
- `Object getObject(int` *`columnIndex`*`)`

  Returns a `String` object in Java 5. Returns a `RowId` object in Java 6.

You can use any of the following `PreparedStatement` methods to set a rowid:

- `setBytes(int parameterIndex, byte[] x)`
- `setString(int parameterIndex, String x)`
- `setRowId(int parameterIndex, RowId x)` (Java 6 only)
- `setObject(int parameterIndex, Object x)`

  Takes a `String` object in Java 5. Takes a `String` or `RowId` object in Java 6.

> **Note:** You cannot use `getBytes()` or `setBytes()` for `ROWID` parameters that are PL/SQL parameters or passthrough parameters (parameters passed to Oracle when using the Oracle In-Memory Database Cache). Use `getString()` and `setString()`, or use `getObject()` and `setObject()` with a `RowId` object (Java 6 only) or `String` object.

An application can specify literal rowid values in SQL statements, such as in `WHERE` clauses, as `CHAR` constants enclosed in single quotes.

Refer to "ROWID data type" and "ROWID specification" in *Oracle TimesTen In-Memory Database SQL Reference* for additional information about rowids and the `ROWID` data type, including usage and lifecycle.

> **Note:** Oracle TimesTen In-Memory Database does not support the PL/SQL type `UROWID`.

Working with synonyms

TimesTen supports private and public synonyms (aliases) for database objects such as tables, views, sequences, and PL/SQL objects. Synonyms are often used for security to mask object names and object owners, or for convenience to simplify SQL statements.

To create a private synonym for table `foo` in your schema:

```
CREATE SYNONYM synfoo FOR foo;
```

To create a public synonym for `foo`:

```
CREATE PUBLIC SYNONYM pubfoo FOR foo;
```

A private synonym exists in the schema of a specific user and shares the same namespace as database objects such as tables, views, and sequences. A private synonym cannot have the same name as a table or other object in the same schema.

A public synonym does not belong to any particular schema, is accessible to all users, and can have the same name as any private object.

To create a synonym you must have the `CREATE SYNONYM` or `CREATE PUBLIC SYNONYM` privilege, as applicable. To use a synonym you must have appropriate privileges to access the underlying object.

For general information about synonyms, see "Understanding synonyms" in *Oracle TimesTen In-Memory Database Operations Guide*. For information about the `CREATE SYNONYM` and `DROP SYNONYM` statements, see "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference*.

Committing or rolling back changes to the database

This section discusses autocommit and manual commits or rollbacks, assuming a JDBC `Connection` object `myconn` and `Statement` object `mystmt`.

> **Note:** All open cursors are closed upon transaction commit or rollback in TimesTen.

Setting autocommit

A TimesTen connection has autocommit enabled by default, but it is recommended that you disable it. You can use the `Connection` method `setAutoCommit()` to enable or disable autocommit.

To disable autocommit:

```
myconn.setAutoCommit(false);
// Report any SQLWarnings on the connection
// See "Reporting errors and warnings" on page 2-31
```

Manually committing or rolling back changes

If autocommit is disabled, you must use the `Connection` method `commit()` to manually commit transactions, or the `rollback()` method to roll back changes. For example:

```
myconn.commit();
```

Or:

```
myconn.rollback();
```

Using COMMIT and ROLLBACK SQL statements

You can prepare and execute `COMMIT` and `ROLLBACK` SQL statements the same way as other SQL statements. Using `COMMIT` and `ROLLBACK` statements has the same effect as using the `Connection` methods `commit()` and `rollback()`. For example:

```
mystmt.execute("COMMIT");
```

Managing multiple threads

> **Note:** On some UNIX platforms it is necessary to set `THREADS_FLAG`, as described in "Set the THREADS_FLAG variable (UNIX only)" in *Oracle TimesTen In-Memory Database Installation Guide*.

The `level4` demo demonstrates the use of multiple threads. Refer to "About the TimesTen Java demos" on page 1-2.

When your application has a direct driver connection to the database, TimesTen functions share stack space with your application. In multithreaded environments, it is important to avoid overrunning the stack allocated to each thread because consequences can result that are unpredictable and difficult to debug. The amount of stack space consumed by TimesTen calls varies depending on the SQL functionality used. Most applications should set thread stack space to at least 16 KB on 32-bit systems and between 34 KB to 72 KB on 64-bit systems.

The amount of stack space allocated for each thread is specified by the operating system when threads are created. On Windows, you can use the TimesTen debug driver and link your application against the Visual C++ debug C library to enable stack probes that raise an identifiable exception if a thread attempts to grow its stack beyond the amount allocated.

Note: In multithreaded applications, a thread that issues requests on different connection handles to the same database may encounter lock conflict with itself. TimesTen resolves these conflicts with lock timeouts.

Java escape syntax and SQL functions

When using SQL in JDBC, pay special attention to Java escape syntax. SQL functions such as `UNISTR` use the backslash (\) character. You should escape the backslash character. For example, using the following SQL syntax in a Java application may not produce the intended results:

```
INSERT INTO table1 SELECT UNISTR('\00E4') FROM dual;
```

Escape the backslash character as follows:

```
INSERT INTO table1 SELECT UNISTR('\\00E4') FROM dual;
```

Using additional TimesTen data management features

Preceding sections discussed key features for managing TimesTen data. This section covers the following additional features:

- Using CALL to execute procedures and functions
- Setting a timeout or threshold for executing SQL statements
- Features for use with IMDB Cache
- Setting up user-specified parallel replication

Using CALL to execute procedures and functions

TimesTen supports each of the following syntax formats from any of its programming interfaces to call PL/SQL procedures (*procname*) or PL/SQL functions (*funcname*) that are standalone or part of a package, or to call TimesTen built-in procedures (*procname*):

```
CALL procname[(argumentlist)]

CALL funcname[(argumentlist)] INTO :returnparam

CALL funcname[(argumentlist)] INTO ?
```

TimesTen JDBC also supports each of the following syntax formats:

```
{ CALL procname[(argumentlist)] }

{ ? = [CALL] funcname[(argumentlist)] }

{ :returnparam = [CALL] funcname[(argumentlist)] }
```

You can execute procedures and functions through the `CallableStatement` interface, with a prepare step first when appropriate (such as when a result set is returned).

The following example calls the TimesTen built-in procedure `ttCkpt`. (Also see Example 2–12 below for a more complete example with JDBC syntax.)

```
CallableStatement.execute("call ttCkpt")
```

The following example calls the TimesTen built-in procedure `ttDataStoreStatus`. A prepare call is used because this procedure produces a result set. (Also see Example 2–13 below for a more complete example with JDBC syntax.)

```
CallableStatement cStmt;
cStmt = conn.prepareCall("call ttDataStoreStatus");
cStmt.execute();
```

These examples call a PL/SQL procedure `myproc` with two parameters:

```
cStmt.execute("{ call myproc(:param1, :param2) }");

cStmt.execute("{ call myproc(?, ?) }");
```

The following shows several ways to call a PL/SQL function `myfunc`:

```
cStmt.execute("CALL myfunc() INTO :retparam");

cStmt.execute("CALL myfunc() INTO ?");

cStmt.execute("{ :retparam = myfunc() }");

cStmt.execute("{ ? = myfunc() }");
```

See "CALL" in *Oracle TimesTen In-Memory Database SQL Reference* for details about `CALL` syntax.

> **Note:** A user's own procedure takes precedence over a TimesTen built-in procedure with the same name.

Example 2–12 Executing a ttCkpt call

This example calls the `ttCkpt` procedure to initiate a fuzzy checkpoint.

```
Connection conn;
CallableStatement cStmt;
.......
cStmt = conn.prepareCall("{ Call ttCkpt }");
cStmt.execute();
conn.commit();           // commit the transaction
```

Be aware that the `ttCkpt` built-in procedure requires `ADMIN` privilege. Refer to "ttCkpt" in *Oracle TimesTen In-Memory Database Reference* for additional information.

Example 2–13 Executing a ttDataStoreStatus call

This example calls the `ttDataStoreStatus` procedure and prints out the returned result set.

For built-in procedures that return results, you can use the `get`*`XXX`*`()` methods of the `ResultSet` interface to retrieve the data, as shown.

Contrary to the advice given in "Working with TimesTen result sets: hints and restrictions" on page 2-10, this example uses a `getString()` call on the `ResultSet` object to retrieve the `Context` field, which is a binary. This is because the output is printed, rather than used for processing. If you do not want to print the `Context` value, you can achieve better performance by using the `getBytes()` method instead.

```
ResultSet rs;

CallableStatement cStmt = conn.prepareCall("{ Call ttDataStoreStatus }");

if (cStmt.execute() == true) {
    rs = cStmt.getResultSet();
    System.out.println("Fetching result set...");
    while (rs.next()) {
      System.out.println("\n Database: " + rs.getString(1));
      System.out.println(" PID: " + rs.getInt(2));
      System.out.println(" Context: " + rs.getString(3));
      System.out.println(" ConType: " + rs.getString(4));
      System.out.println(" memoryID: " + rs.getString(5));
      }
    rs.close();
  }
cStmt.close();
```

Setting a timeout or threshold for executing SQL statements

TimesTen offers two ways to limit the time for SQL statements to execute, applying to any `execute()`, `executeBatch()`, `executeQuery()`, `executeUpdate()`, or `next()` call.

- Setting a timeout value for SQL statements
- Setting a threshold value for SQL statements

The former is to set a timeout, where if the timeout duration is reached, the statement stops executing and an error is thrown. The latter is to set a threshold, where if the threshold is reached, an SNMP trap is thrown but execution continues.

Setting a timeout value for SQL statements

In TimesTen you can set the `SqlQueryTimeout` general connection attribute to specify the timeout period (in seconds) for any connection, and hence any statement. If you set `SqlQueryTimeout` in the DSN specification, its value becomes the default value for all subsequent connections to the database. Despite the name, this timeout value applies to any executable SQL statement, not just queries.

For a particular statement, you can override the `SqlQueryTimeout` setting by calling the `Statement` method `setQueryTimeout()`.

The query timeout limit has effect only when the SQL statement is actively executing. A timeout does not occur during the commit or rollback phase of an operation. For those transactions that execute a large number of `UPDATE`, `DELETE`, or `INSERT` statements, the commit or rollback phases may take a long time to complete. During that time the timeout value is ignored.

Note: If both a lock-wait and a `SqlQueryTimeout` are specified, the lesser of the two values causes a timeout first. Regarding lock timeouts, in *Oracle TimesTen In-Memory Database Reference* you can refer to information about the `ttLockWait` built-in procedure in "ttLockWait" and about the `LockWait` general connection attribute in "LockWait". Or refer to "Check for deadlocks and timeouts" in the *Oracle TimesTen In-Memory Database Troubleshooting Procedures Guide*.

Setting a threshold value for SQL statements

You can configure TimesTen to write a warning to the support log and throw an SNMP trap when the execution of a SQL statement exceeds a specified time duration, in seconds. Execution continues and is not affected by the threshold.

The name of the SNMP trap is `ttQueryThresholdWarnTrap`. See *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps* for information about configuring SNMP traps.

Despite the name, this threshold applies to any JDBC call executing a SQL statement, not just queries.

By default, the application obtains the threshold value from the `QueryThreshold` general connection attribute setting. You can override the threshold for a JDBC `Connection` object by including the `QueryThreshold` attribute in the connection URL for the database. For example, to set `QueryThreshold` to a value of 5 seconds for the `myDSN` database:

```
jdbc:timesten:direct:dsn=myDSN;QueryThreshold=5
```

You can also use the `setQueryTimeThreshold()` method of a `TimesTenStatement` object to set the threshold. This overrides the connection attribute setting and the `Connection` object setting.

You can retrieve the current threshold value by using the `getQueryTimeThreshold()` method of the `TimesTenStatement` object.

Features for use with IMDB Cache

This section discusses features related to the use of IMDB Cache:

- Setting temporary passthrough level with the ttOptSetFlag built-in procedure
- Managing cache groups

Note: The `OraclePassword` attribute maps to the Oracle password. You can use the `TimesTenDataSource` method `setOraclePassword()` to set the Oracle password. See "Connect to the database" on page 2-6 for an example.

Setting temporary passthrough level with the ttOptSetFlag built-in procedure

TimesTen provides the `ttOptSetFlag` built-in procedure for setting various flags, including the `PassThrough` flag to temporarily set the passthrough level. You can use `ttOptSetFlag` to set `PassThrough` in a JDBC application as in the following sample statement, which sets the passthrough level to 1. The setting affects all statements that are prepared until the end of the transaction.

```
pstmt = conn.prepareStatement("call ttoptsetflag('PassThrough', 1)");
```

The example that follows includes samples of code that accomplish these steps:

1. Creation of a prepared statement (a `PreparedStatement` instance `thePassThroughStatement`) that calls `ttOptSetFlag` using a bind parameter for passthrough level.
2. Definition of a method `setPassthrough()` that takes a specified passthrough setting, binds it to the prepared statement, then executes the prepared statement to call `ttOptSetFlag` to set the passthrough level.

```
  thePassThroughStatement =
         theConnection.prepareStatement("call ttoptsetflag('PassThrough', ?)");
...
  private void setPassthrough(int level) throws SQLException{
    thePassThroughStatement.setInt(1, level);
    thePassThroughStatement.execute();
  }
```

Also see "ttOptSetFlag" in *Oracle TimesTen In-Memory Database Reference* for more information about that built-in procedure, and "Setting a passthrough level" in *Oracle In-Memory Database Cache User's Guide* for information about the meaning and effect of each passthrough level.

Managing cache groups

In TimesTen, following the execution of a `FLUSH CACHE GROUP`, `LOAD CACHE GROUP`, `REFRESH CACHE GROUP`, or `UNLOAD CACHE GROUP` statement, the `Statement` method `getUpdateCount()` returns the number of cache instances that were flushed, loaded, refreshed, or unloaded.

For related information, see "Determining the number of cache instances affected by an operation" in *Oracle In-Memory Database Cache User's Guide*.

Setting up user-specified parallel replication

For applications that have very predictable transactional dependencies and do not require the commit order on the replica database to be the same as that on the originating database, TimesTen supports *parallel replication*. This feature allows replication of multiple user-specified *tracks* of transactions in parallel. See "Increasing replication throughput for other replication schemes" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for general information about parallel replication.

User-specified parallel replication is enabled through the TimesTen data store attributes `ReplicationParallelism` and `ReplicationApplyOrdering`, as described in "Data store attributes" in *Oracle TimesTen In-Memory Database Reference*. The track number of transactions for a connection can be specified through the TimesTen general connection attribute `ReplicationTrack`, the `ALTER SESSION` parameter `REPLICATION_TRACK`, or in JDBC through the following `TimesTenConnection` method:

- `void setReplicationTrack(int` *`track`*`)`

`TimesTenConnection` also has the corresponding getter method:

- `int getReplicationTrack()`

Note: The track number setting will hold for the lifetime of the connection, unless it is specifically reset.

To find the track number that is in use, you can call the `TimesTenConnection` method `getReplicationTrack()` or call the TimesTen built-in procedure `ttConfiguration`, which returns current TimesTen attribute settings, including `ReplicationTrack`.

Considering TimesTen features for access control

TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. You can refer to "Managing Access Control" in *Oracle TimesTen In-Memory Database Operations Guide* for introductory information about TimesTen access control.

This section introduces access control as it relates to SQL operations, database connections, and JMS/XLA.

For any query, SQL DML statement, or SQL DDL statement discussed in this document or used in an example, it is assumed that the user has appropriate privileges to execute the statement. For example, a `SELECT` statement on a table requires ownership of the table, `SELECT` privilege granted for the table, or the `SELECT ANY TABLE` system privilege. Similarly, any DML statement requires table ownership, the applicable DML privilege (such as `UPDATE`) granted for the table, or the applicable `ANY TABLE` privilege (such as `UPDATE ANY TABLE`).

For DDL statements, `CREATE TABLE` requires the `CREATE TABLE` privilege in the user's schema, or `CREATE ANY TABLE` in any other schema. `ALTER TABLE` requires ownership or the `ALTER ANY TABLE` system privilege. `DROP TABLE` requires ownership or the `DROP ANY TABLE` system privilege. There are no object-level `ALTER` or `DROP` privileges.

Refer to "SQL Statements" in *Oracle TimesTen In-Memory Database SQL Reference* for a list of access control privileges and the privilege required for any given SQL statement.

Privileges are granted through the SQL statement `GRANT` and revoked through the statement `REVOKE`. Some privileges are automatically granted to all users through the `PUBLIC` role, of which all users are a member. Refer to "The PUBLIC role" in *Oracle TimesTen In-Memory Database SQL Reference* for information about this role.

In addition, access control affects the following topics covered in this document:

- Connecting to a database. Refer to "Access control for connections" on page 2-8.
- Setting connection attributes. Refer to "Create a connection URL for the database and specify connection attributes" on page 2-6.
- Configuring and managing JMS/XLA. Refer to "Access control impact on XLA" on page 3-8.

Notes:

- Access control cannot be disabled.
- Access control privileges are checked both when SQL is prepared and when it is executed in the database, with most of the performance cost coming at prepare time.

Handling errors

This section discusses how to check for, identify and handle errors in a TimesTen Java application.

For a list of the errors that TimesTen returns and what to do if the error is encountered, see "Warnings and Errors" in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps*.

This section includes the following topics.

About fatal errors, non-fatal errors, and warnings

TimesTen can return a fatal error, a non-fatal error, or a warning.

Handling fatal errors

Fatal errors make the database inaccessible until it can be recovered. When a fatal error occurs, all database connections are required to disconnect. No further operations may complete. Fatal errors are indicated by TimesTen error codes 846 and 994. Error handling for these errors should be different from standard error handling. In particular, the code should roll back the transaction and, to avoid out-of-memory conditions, disconnect from the database. Shared memory from the old TimesTen instance will not be freed until all active connections at the time of the error have disconnected.

When fatal errors occur, TimesTen performs the full cleanup and recovery procedure:

- Every connection to the database is invalidated, a new memory segment is allocated and applications are required to disconnect.
- The database is recovered from the checkpoint and transaction log files upon the first subsequent initial connection.
  - The recovered database reflects the state of all durably committed transactions and possibly some transactions that were committed non-durably.
  - No uncommitted or rolled back transactions are reflected.

If no checkpoint or transaction log files exist and the `AutoCreate` attribute is set, TimesTen creates an empty database.

Handling non-fatal errors

Non-fatal errors include simple errors such as an `INSERT` statement that violates unique constraints. This category also includes some classes of application and process failures.

TimesTen returns non-fatal errors through the normal error-handling process and requires the application to check for and identify them.

When a database is affected by a non-fatal error, an error may be returned and the application should take appropriate action. In some cases, such as with a process failure, an error cannot be returned, so TimesTen automatically rolls back the transactions of the failed process.

An application can handle non-fatal errors by modifying its actions or, in some cases, by rolling back one or more offending transactions, as described in "Rolling back failed transactions" on page 2-33.

> **Note:** If a `ResultSet`, `Statement`, `PreparedStatement`, `CallableStatement` or `Connection` operation results in a database error, it is a good practice to call the `close()` method for that object.

About warnings

TimesTen returns warnings when something unexpected occurs that you may want to know about. Here are some examples of events that cause TimesTen to issue a warning:

- A checkpoint failure
- Use of a deprecated TimesTen feature
- Truncation of some data
- Execution of a recovery process upon connect

You should always include code that checks for warnings, as they can indicate application problems.

Reporting errors and warnings

You should check for and report all errors and warnings that can be returned on every call. This saves considerable time and effort during development and debugging. A `SQLException` object is generated if there are one or more database access errors and a `SQLWarning` object is generated if there are one or more warning messages. A single call may return multiple errors or warnings or both, so your application should report all errors or warnings in the returned `SQLException` or `SQLWarning` objects.

Multiple errors or warnings are returned in linked chains of `SQLException` or `SQLWarning` objects. Example 2–14 and Example 2–15 demonstrate how you might iterate through the lists of returned `SQLException` and `SQLWarning` objects to report all of the errors and warnings, respectively.

Example 2–14 Printing exceptions

This method prints out the content of all exceptions in the linked `SQLException` objects.

```
static int reportSQLExceptions(SQLException ex)
  {
    int errCount = 0;

    if (ex != null) {
      errStream.println("\n--- SQLException caught ---");
      ex.printStackTrace();

      while (ex != null) {
        errStream.println("SQL State: " + ex.getSQLState());
        errStream.println("Message: " + ex.getMessage());
        errStream.println("Error Code: " + ex.getErrorCode());
        errCount ++;
        ex = ex.getNextException();
        errStream.println();
```

```
      }
    }

    return errCount;
}
```

Example 2–15 Printing warnings

This method prints out the content of all warning in the linked `SQLWarning` objects.

```
static int reportSQLWarnings(SQLWarning wn)
{
    int warnCount = 0;

    while (wn != null) {
      errStream.println("\n--- SQL Warning ---");
      errStream.println("SQL State: " + wn.getSQLState());
      errStream.println("Message: " + wn.getMessage());
      errStream.println("Error Code: " + wn.getErrorCode());

      // is this a SQLWarning object or a DataTruncation object?
      if (wn instanceof DataTruncation) {
        DataTruncation trn = (DataTruncation) wn;
        errStream.println("Truncation error in column: " +
          trn.getIndex());
      }

      warnCount++;
      wn = wn.getNextWarning();
      errStream.println();
    }

    return warnCount;
}
```

Catching and responding to specific errors

In some situations it may be desirable to respond to a specific SQL state or TimesTen error code. You can use the `SQLException` method `getSQLState()` to return the SQL99 state error string, and `getErrorCode()` to return TimesTen error codes, as shown in Example 2–16.

Also refer to the entry for `TimesTenVendorCode` in *Oracle TimesTen In-Memory Database JDBC Extensions Java API Reference* for error information.

Example 2–16 Catching an error

The TimesTen demos require that you load the demo schema before they are executed. The following `catch` statement alerts the user that `appuser` has not been loaded or has not been refreshed by detecting ODBC error `S0002` and TimesTen error 907:

```
catch (SQLException ex) {
  if (ex.getSQLState().equalsIgnoreCase("S0002")) {
    errStream.println("\nError: The table appuser.customer " +
      "does not exist.\n\t Please reinitialize the database.");
  } else if (ex.getErrorCode() == 907) {
    errStream.println("\nError: Attempting to insert a row " +
      "with a duplicate primary key.\n\tPlease reinitialize the database.");
}
```

You can use the `TimesTenVendorCode` interface to detect the errors by their name, rather than their number.

Consider this example:

```
ex.getErrorCode() == com.timesten.jdbc.TimesTenVendorCode.TT_ERR_KEYEXISTS
```

The following is equivalent:

```
ex.getErrorCode() == 907
```

Rolling back failed transactions

In some situations, such as recovering from a deadlock or timeout condition, you may want to explicitly roll back the transaction using the `Connection` method `rollback()`, as in the following example.

Example 2–17 Rolling back a transaction

```
try {
  if (conn != null && !conn.isClosed()) {
    // Rollback any transactions in case of errors
      if (retcode != 0) {
        try {
          System.out.println("\nEncountered error. Rolling back transaction");
          conn.rollback();
        } catch (SQLException ex) {
          reportSQLExceptions(ex);
        }
      }
   }

    System.out.println("\nClosing the connection\n");
    conn.close();
}
```

The `XACT_ROLLBACKS` column of the `SYS.MONITOR` table indicates the number of transactions that were rolled back. Refer to "SYS.MONITOR" in *Oracle TimesTen In-Memory Database System Tables and Limits Reference* for additional information.

A transaction rollback consumes resources and the entire transaction is in effect wasted. To avoid unnecessary rollbacks, design your application to avoid contention and check the application or input data for potential errors before submitting it.

Note: If your application fails in the middle of an active transaction, TimesTen automatically rolls back the transaction.

JDBC support for automatic client failover

Automatic client failover, used in High Availability scenarios when failure of a TimesTen node results in failover (transfer) to an alternate node, automatically reconnects applications to the new node. TimesTen provides features that allow applications to be alerted when this happens, so they can take any appropriate action.

This section discusses TimesTen JDBC extensions related to automatic client failover, covering the following topics:

- Features and functionality of JDBC support for automatic client failover
- Synchronous detection of automatic client failover

- Asynchronous detection of automatic client failover

> **Note:** Automatic client failover applies only to client/server mode. The functionality described here does not apply to a direct connection.

Automatic client failover is complementary to Oracle Clusterware in situations where Oracle Clusterware is used, though the two features are not dependent on each other.

You can refer to "Automatic client failover" in *Oracle TimesTen In-Memory Database C Developer's Guide* for related information. You can also refer to "Using Oracle Clusterware to Manage Active Standby Pairs" in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* for information about Oracle Clusterware.

Features and functionality of JDBC support for automatic client failover

This section discusses general TimesTen JDBC features related to client failover, and functionality relating specifically to pooled connections.

General Client Failover Features

TimesTen JDBC support for automatic client failover provides two mechanisms for detecting a failover:

- *Synchronous detection*, through a SQL exception. After an automatic client failover, JDBC objects created on the failed connection—such as statements, prepared statements, callable statements, and result sets—can no longer be used. A Java SQL exception is thrown if an application attempts to access any such object. By examining the SQL state and error code of the exception, you can determine whether the exception is the result of a failover situation.
- *Asynchronous detection*, through an event listener. An application can register a user-defined client failover event listener, which will be notified of each event that occurs during the process of a failover.

TimesTen JDBC provides the following features, in package `com.timesten.jdbc`, to support automatic client failover:

- The `ClientFailoverEvent` class. This class is used to represent events that occur during a client failover: begin, end, abort, or retry.
- The `ClientFailoverEventListener` interface. An application interested in client failover events must include a class that implements this interface, which is the mechanism to listen for client failover events. At runtime, the application must register `ClientFailoverEventListener` instances through the TimesTen connection (see immediately below).
- New methods in the `TimesTenConnection` interface. This interface specifies the methods `addConnectionEventListener()` and `removeConnectionEventListener()` to register or remove, respectively, a client failover event listener.
- A new constant, `TT_ERR_FAILOVERINVALIDATION`, in the `TimesTenVendorCode` interface. This enables you to identify an event as a failover event.

Client failover features for pooled connections

TimesTen recommends that applications using pooled connections (`javax.sql.PooledConnection`) or connection pool data sources

(javax.sql.ConnectionPoolDataSource) use the synchronous mechanism noted previously to handle stale objects on the failed connection. Java EE application servers manage pooled connections, so applications are not able to listen for events on pooled connections. And application servers would not implement and register an instance of ClientFailoverEventListener, that being a TimesTen extension.

Synchronous detection of automatic client failover

If, in a failover situation, an application attempts to use objects created on the failed connection, then JDBC will throw a SQL exception. The vendor-specific exception code will be set to TimesTenVendorCode.TT_ERR_FAILOVERINVALIDATION.

Detecting a failover through this mechanism is referred to as synchronous detection. The following example demonstrates this.

Example 2–18 Synchronous detection of automatic client failover

```
try {
   // ...
   // Execute a query on a previously prepared statement.
   ResultSet theResultSet = theStatement.executeQuery("select * from dual");
   // ...

} catch (SQLException sqlex) {
   sqlex.printStackTrace();
   if (sqlex.getErrorCode() == TimesTenVendorCode.TT_ERR_FAILOVERINVALIDATION) {
   // Automatic client failover has taken place; discontinue use of this object.
   }
}
```

Asynchronous detection of automatic client failover

Asynchronous failover detection requires an application to implement a client failover event listener and register an instance of it on the TimesTen connection. This section describes the steps involved:

1. Implement a client failover event listener
2. Register the client failover listener instance
3. Remove the client failover listener instance

Implement a client failover event listener

TimesTen JDBC provides the com.timesten.jdbc.ClientFailoverEventListener interface for use in listening for events, highlighted by the following method:

- void notify(ClientFailoverEvent *event*)

To use asynchronous failover detection, you must create a class that implements this interface, then register an instance of the class at runtime on the TimesTen connection (discussed shortly).

When a failover event occurs, TimesTen calls the notify() method of the listener instance you registered, providing a ClientFailoverEvent instance that you can then examine for information about the event.

The following example shows the basic form of a ClientFailoverEventListener implementation.

Example 2–19 Asynchronous detection of automatic client failover

```
private class MyCFListener implements ClientFailoverEventListener {
   // Skeletal example
   /* Applications can build state system to track states during failover.
      You may want to add methods that talks about readiness of this Connection
      for processing.
   */
   public void notify(ClientFailoverEvent event) {

      // Process connection failover type
      switch(event.getTheFailoverType()) {
      case TT_FO_CONNECTION:
         // Process session fail over
         System.out.println("This should be a connection failover type " +
                            event.getTheFailoverType());
         break;

      default:
         break;
      }

      // Process connection failover events
      switch(event.getTheFailoverEvent()) {
      case BEGIN:
         System.out.println("This should be a BEGIN event " +
                            event.getTheFailoverEvent());

         /* Applications cannot use Statement, PreparedStatement, ResultSet,
            etc. created on the failed Connection any longer.
         */
         // ...
         break;

      case END:
         System.out.println("This should be an END event " +
                            event.getTheFailoverEvent());

         /* Applications may want to re-create Statement and PreparedStatement
            objects at this point as needed.
         */

         break;

      case ABORT:
         System.out.println("This should be an ABORT event " +
                            event.getTheFailoverEvent());
         break;

      case ERROR:
         System.out.println("This should be an ERROR event " +
                            event.getTheFailoverEvent());
         break;

      default:
         break;
      }
   }
}
```

The `event.getTheFailoverType()` call returns an instance of the nested class `ClientFailoverEvent.FailoverType`, which is an enumeration type. In TimesTen, the only supported value is `TT_FO_CONNECTION`, indicating a connection failover.

The `event.getTheFailoverEvent()` call returns an instance of the nested class `ClientFailoverEvent.FailoverEvent`, which is an enumeration type where the value can be one of the following:

- `BEGIN` if the client failover has begun
- `END` if the client failover has completed successfully
- `ERROR` if the client failover failed but will be retried
- `ABORT` if the client failover has aborted

Register the client failover listener instance

At runtime you must register an instance of your failover event listener class with the TimesTen connection object, so that TimesTen will be able to call the `notify()` method of the listener class as needed for failover events.

`TimesTenConnection` provides the following method for this:

- `void addConnectionEventListener (ClientFailoverEventListener` *`listener`*`)`

Create an instance of your listener class, then register it using this method. The following example establishes the connection and registers the listener. Assume `theDsn` is the JDBC URL for a TimesTen Client/Server database and `theCFListener` is an instance of your failover event listener class.

Example 2–20 Registering the client failover listener

```
try {

   // Assume this is a client/server connection; register for conn failover.
   Class.forName("com.timesten.jdbc.TimesTenClientDriver");
   String url = "jdbc:timesten:client:" + theDsn;
   theConnection = (TimesTenConnection)DriverManager.getConnection(url);
   theConnection.addConnectionEventListener(theCFListener);
   // ...
   /* Additional logic goes here; connection failover listener will be
      called if there is a fail over.
   */
   // ...
}
catch (ClassNotFoundException cnfex) {
   cnfex.printStackTrace();
}
catch (SQLException sqlex) {
   sqlex.printStackTrace();
}
```

Remove the client failover listener instance

The `TimesTenConnection` interface defines the following method to deregister a failover event listener instance:

- `void removeConnectionEventListener (ClientFailoverEventListener` *`listener`*`)`

3

Using JMS/XLA for Event Management

You can use the TimesTen JMS/XLA API (JMS/XLA) to monitor TimesTen for changes to specified tables in a local database and receive real-time notification of these changes. One of the purposes of JMS/XLA is to provide a high-performance, asynchronous alternative to triggers.

You can also use JMS/XLA to build a custom data replication solution, if the TimesTen replication solutions described in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* do not meet your needs.

JMS/XLA implements Java Message Service (JMS) interfaces to make the functionality of the TimesTen Transaction Log API (XLA) available to Java applications. JMS information and resources are available at the following location:

`http://java.sun.com/products/jms/docs.html`

In addition, the standard JMS API documentation is installed with the Oracle TimesTen In-Memory Database at the following location:

`install_dir``/3rdparty/jms1.1/doc/api/index.html`

For information about tuning TimesTen JMS/XLA applications for improved performance, see "Tuning JMS/XLA applications" on page 5-4.

This chapter includes the following topics:

JMS/XLA concepts

Java applications can use the JMS/XLA API to receive event notifications from TimesTen. JMS/XLA uses the JMS publish-subscribe interface to provide access to XLA updates.

You subscribe to updates by establishing a JMS `Session` that provides a connection to XLA and creating a durable subscriber (`TopicSubscriber`). You can receive and process messages synchronously through the subscriber, or you can implement a listener (`MessageListener`) to process the updates asynchronously.

JMS/XLA is designed for applications that want to monitor a local database. TimesTen and the application receiving the notifications must reside on the same system.

> **Note:** The JMS/XLA API supports persistent-mode XLA. In this mode, XLA obtains update records directly from the transaction log buffer or transaction log files, so the records are available until they are read. Persistent-mode XLA also allows multiple readers to access transaction log updates simultaneously.

This section includes the following topics:

- How XLA reads records from the transaction log
- XLA and materialized views
- XLA bookmarks
- JMS/XLA configuration file and topics
- XLA updates
- XLA acknowledgment modes
- Access control impact on XLA

How XLA reads records from the transaction log

As applications modify a database, TimesTen generates transaction log records that describe the changes made to the data and other events such as transaction commits.

New transaction log records are always written to the end of the transaction log buffer as they are generated. Transaction log records are periodically flushed in batches from the log buffer in memory to transaction log files on disk.

Applications can use XLA to monitor the transaction log for changes to the database. XLA reads through the transaction log, filters the log records, and delivers XLA applications with a list of transaction records that contain the changes to the tables and columns of interest.

XLA sorts the records into discrete transactions. If multiple applications are updating the database simultaneously, transaction log records from the different applications will be interleaved in the transaction log.

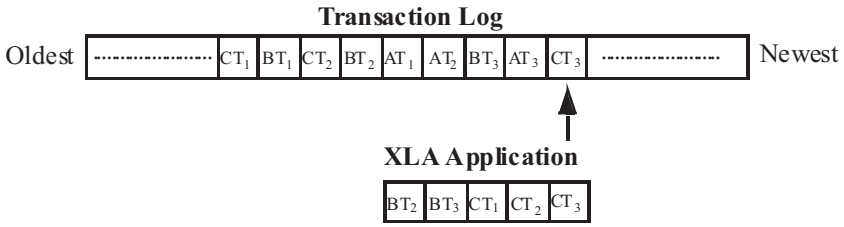
XLA transparently extracts all transaction log records associated with a particular transaction and delivers them in a contiguous list to the application.

Only the records for committed transactions are returned. They are returned in the order in which their final commit record appears in the transaction log. XLA filters out records associated with changes to the database that have not yet committed.

If a change is made but then rolled back, XLA does not deliver the records for the aborted transaction to the application.

Consider the example transaction log illustrated in Figure 3–1 and Example 3–1 that follow, which illustrate most of these basic XLA concepts.

Figure 3–1 Records extracted from the transaction log



Example 3–1 Reading transaction log records

In this example, the transaction log contains the following records:

CT1 - Application C updates row 1 of table W with value 7.7.
BT1 - Application B updates row 3 of table X with value 2.
CT2 - Application C updates row 9 of table W with value 5.6.
BT2 - Application B updates row 2 of table Y with value "XYZ".
AT1 - Application A updates row 1 of table Z with value 3.
AT2 - Application A updates row 3 of table Z with value 4.
BT3 - Application B commits its transaction.
AT3 - Application A rolls back its transaction.
CT3 - Application C commits its transaction.

An XLA application that is set up to detect changes to tables W, Y, and Z would see the following:

BT2 and BT3 - Update row 2 of table Y with value "XYZ" and commit.
CT1 - Update row 1 of table W with value 7.7.
CT2 and CT3 - Update row 9 of table W with value 5.6 and commit.

This example demonstrates the following:

- Transaction records for application B and application C all appear.
- Though the records for application C begin to appear in the transaction log before those for application B, the commit for application B (BT3) appears in the transaction log before the commit for application C (CT3). As a result, the records for application B are returned to the XLA application ahead of those for application C.
- The application B update to table X (BT1) is not presented because XLA is not set up to detect changes to table X.
- The application A updates to table Z (AT1 and AT2) are never presented because it did not commit and was rolled back (AT3).

XLA and materialized views

You can use XLA to track changes to both tables and materialized views. A materialized view provides a single source from which you can track changes to selected rows and columns in multiple detail tables. Without a materialized view, the XLA application would have to monitor and filter the update records from all of the detail tables, including records reflecting updates to rows and columns of no interest to the application.

In general, there are no operational differences between the XLA mechanisms used to track changes to a table or a materialized view. However, for asynchronous materialized views, be aware that the order of XLA notifications for an asynchronous

view is not necessarily the same as it would be for the associated detail tables, or the same as it would be for a synchronous view. For example, if there are two inserts to a detail table, they may be done in the opposite order in the asynchronous materialized view. Furthermore, updates may be treated as a delete followed by an insert, and multiple operations (such as multiple inserts or multiple deletes) may be combined. Applications that depend on ordering should not use asynchronous materialized views.

XLA bookmarks

An XLA bookmark marks the read position of an XLA subscriber application in the transaction log. Bookmarks facilitate durable subscriptions, enabling an application to disconnect from a topic and then reconnect to continue receiving updates where it left off.

How bookmarks work

When you create a message consumer for XLA, you always use a durable `TopicSubscriber`. The subscription identifier you specify when you create the subscriber is used as the XLA bookmark name. When you use the `ttXlaSubscribe` and `ttXlaUnsubscribe` built-in procedures through JDBC to start and stop XLA publishing for a table, you explicitly specify the name of the bookmark to be used.

Bookmarks are reset to the last read position whenever an acknowledgment is received. For more information about how update messages are acknowledged, see the "XLA acknowledgment modes" on page 3-7.

You can remove a durable subscription by calling `unsubscribe()` on the JMS `Session` object. This deletes the corresponding XLA bookmark and forces a new subscription to be created when you reconnect. For more information see "Deleting bookmarks" on page 3-12.

A bookmark subscription cannot be altered when it is in use. To alter a subscription, you must close the message consumer, alter the subscription using `ttXlaSubscribe` and `ttXlaUnsubscribe`, and open the message consumer.

Note: You can also use the `ttXlaBookmarkCreate` TimesTen built-in procedure to create bookmarks. See "ttXlaBookmarkCreate" in *Oracle TimesTen In-Memory Database Reference* for information about that function.

Replicated bookmarks

If you are using an active standby pair replication scheme, you have the option of using *replicated bookmarks*, according to the `replicatedBookmark` attribute of the `<topic>` element in the `jmsxla.xml` file as discussed in "JMS/XLA configuration file and topics" on page 3-5. For a replicated bookmark, operations on the bookmark are replicated to the standby database as appropriate, assuming there is suitable write privilege for the standby. This allows more efficient recovery of your bookmark positions if a failover occurs.

You can only read and acknowledge a replicated bookmark in the active database. Each time you acknowledge a replicated bookmark, the acknowledge operation is asynchronously replicated to the standby database.

Note: Alternatively, if you use `ttXlaBookmarkCreate` to create a bookmark, that function has a bit you can set to specify a replicated bookmark.

Be aware of the following usage notes:

- The position of the bookmark in the standby database will be very close to that of the bookmark in the active database; however, because the replication of acknowledge operations is asynchronous, you may see a small window of duplicate updates when there is a failover, depending on how often acknowledge operations are performed.
- If replicated bookmarks exist at the time you enable the active standby pair scheme, the bookmarks will automatically be added to the replication scheme.
- It is permissible to drop the active standby pair scheme while replicated bookmarks exist. The bookmarks will cease to be replicated at that point.
- You cannot delete replicated bookmarks while the replication agent is running.

JMS/XLA configuration file and topics

To connect to XLA, you establish a connection to a JMS `Topic` object that corresponds to a particular database. The JMS/XLA configuration file provides the mapping between topic names and databases.

You can specify a replicated bookmark by setting `replicatedBookmark="yes"` in the `<topic>` element when you specify the topic. The default setting is `"no"`. Also see "XLA bookmarks" on page 3-4.

By default, JMS/XLA looks for a configuration file named `jmsxla.xml` in the current working directory. If you want to use another name or location for the file, you must specify it as part of the environment variable in the `InitialContext` class and add the location to the classpath.

Example 3–2 Specifying the JMS/XLA configuration file

The following code specifies the configuration file as part of the environment variable in the `InitialContext` class.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
  "com.timesten.dataserver.jmsxla.SimpleInitialContextFactory");
env.put(XlaConstants.CONFIG_FILE_NAME, "/newlocation.xml");
InitialContext ic = new InitialContext(env);
```

The JMS/XLA API uses the class loader to locate the JMS/XLA configuration file if `XlaConstants.CONFIG_FILE_NAME` is set. In this example, the JMS/XLA API searches for the `newlocation.xml` file in the top directory in both the location specified in the `CLASSPATH` environment variable and in the JAR files specified in the `CLASSPATH` variable.

The JMS/XLA configuration file can also be located in subdirectories, as follows:

```
env.put(XlaConstants.CONFIG_FILE_NAME,
"/com/mycompany/myapplication/deepinside.xml");
```

In this case, the JMS/XLA API searches for the `deepinside.xml` file in the `com/mycompany/myapplication` subdirectory in both the location specified in the

CLASSPATH environment variable and in the JAR files specified in the CLASSPATH variable.

The JMS/XLA API uses the first configuration file that it finds.

Example 3–3 Defining a topic in the configuration file

A topic definition in the configuration file consists of a name, a connection string, and a prefetch value that specifies how many updates to retrieve at a time.

For example, the configuration file shown here maps the DemoDataStore topic to the TestDB DSN:

```
<xlaconfig>
  <topics>
    <topic name="DemoDataStore"
      connectionString="DSN=TestDB"
      xlaPrefetch="100" />
  </topics>
</xlaconfig>
```

Example 3–4 Defining a topic to use replicated bookmarks

A topic definition can also specify whether a replicated bookmark should be used. The following repeats the preceding example, but with a replicated bookmark.

```
<xlaconfig>
  <topics>
    <topic name="DemoDataStore"
      connectionString="DSN=TestDB"
      xlaPrefetch="100" replicatedBookmark="yes" />
  </topics>
</xlaconfig>
```

XLA updates

Applications receive XLA updates as JMS MapMessage objects. The MapMessage contains a set of typed name and value pairs that correspond to the fields in an XLA update header.

You can access the message fields using the MapMessage getter methods. The getMapNames() method returns an Enumeration object that contains the names of all of the fields in the message. You can retrieve individual fields from the message by name. All reserved field names begin with two underscores, for example __TYPE.

All update messages have a __TYPE field that indicates what type of update the message contains. The types are specified as integer values. As a convenience, you can use the constants defined in com.timesten.dataserver.jmsxla.XlaConstants to compare against the integer types. The supported types are described in Table 3–1.

Table 3–1 XLA update types

Update type	Description
INSERT	A row has been added.
UPDATE	A row has been modified.
DELETE	A row has been removed.
COMMIT_ONLY	A transaction has been committed.
CREATE_TABLE	A table has been created.

Table 3–1 (Cont.) XLA update types

Update type	Description
DROP_TABLE	A table has been dropped.
CREATE_INDEX	An index has been created.
DROP_INDEX	An index has been dropped.
ADD_COLUMNS	New columns have been added to the table.
DROP_COLUMNS	Columns have been removed from the table.
CREATE_VIEW	A materialized view has been created.
DROP_VIEW	A materialized view has been dropped.
CREATE_SEQ	A sequence has been created.
DROP_SEQ	A sequence has been dropped.
CREATE_SYNONYM	A synonym has been created.
DROP_SYNONYM	A synonym has been dropped.
TRUNCATE	The table has been truncated and all rows in the table have been deleted.

For more information about the contents of an XLA update message, see "JMS/XLA MapMessage contents" on page 6-1.

XLA acknowledgment modes

The XLA acknowledgment mechanism is designed to ensure that an application has not only received a message, but has successfully processed it. Acknowledging an update permanently resets the application's XLA bookmark to the last record that was read. This prevents previously returned records from being reread, ensuring that an application does not receive previously acknowledged records if the bookmark is reused when an application reconnects to XLA.

JMS/XLA can automatically acknowledge XLA update messages, or applications can choose to acknowledge messages explicitly. You specify how updates are to be acknowledged when you create the `Session` object.

JMS/XLA supports three acknowledgment modes:

- `AUTO_ACKNOWLEDGE`: In this mode, updates are automatically acknowledged as you receive them. Each message is delivered only once. Duplicate messages will not be sent, so messages might be lost if there is an application failure. Messages are always delivered and acknowledged individually, so JMS/XLA does not prefetch multiple records. The `xlaprefetch` attribute in the topic is ignored.
- `DUPS_OK_ACKNOWLEDGE`: In this mode, updates are automatically acknowledged, but duplicate messages might be delivered when there is an application failure. JMS/XLA prefetches records according to the `xlaprefetch` attribute specified for the topic and sends an acknowledgment when the last record in a prefetched block is read. If the application fails before reading all of the prefetched records, all of the records in the block are presented to the application it restarts.
- `CLIENT_ACKNOWLEDGE`: In this mode, applications are responsible for acknowledging receipt of update messages by calling `acknowledge()` on the `MapMessage`. JMS/XLA prefetches records according to the `xlaprefetch` attribute specified for the topic.

Prefetching updates

Prefetching multiple update records at a time is more efficient than obtaining each update record from XLA individually. Because updates are not prefetched when you use `AUTO_ACKNOWLEDGE` mode, it can be slower than the other modes. If possible, you should design the application to tolerate duplicate updates so you can use `DUPS_OK_ACKNOWLEDGE`, or explicitly acknowledge updates. Explicitly acknowledging updates usually yields the best performance, as long as you can avoid acknowledging each message individually.

Acknowledging updates

To explicitly acknowledge an XLA update, call `acknowledge()` on the update message. Acknowledging a message implicitly acknowledges all previous messages. Typically, you receive and process multiple update messages between acknowledgments. If you are using the `CLIENT_ACKNOWLEDGE` mode and intend to reuse a durable subscription in the future, you should call `acknowledge()` to reset the bookmark to the last-read position before exiting.

Access control impact on XLA

"Considering TimesTen features for access control" on page 2-29 provides a brief overview of how TimesTen access control affects operations in the database. Access control includes impact on XLA, as follows:

- Any XLA functionality requires the system privilege `XLA`. This includes connecting to TimesTen as an XLA reader and executing the TimesTen XLA built-in procedures `ttXlaBookmarkCreate`, `ttXlaBookmarkDelete`, `ttXlaSubscribe`, and `ttXlaUnsubscribe`, all of which are documented in "Built-In Procedures" in *Oracle TimesTen In-Memory Database Reference*.
- A user with the `XLA` privilege has capabilities equivalent to the `SELECT ANY TABLE` and `SELECT ANY SEQUENCE` system privileges.

JMS/XLA and Oracle GDK dependency

The JMS/XLA API uses `orai18n.jar`, part of the Oracle Globalization Development Kit (GDK) for translating from the database character set specified by the `DatabaseCharacterSet` attribute to UTF-16 encoding. The JMS/XLA API supports a specific version of the GDK with each TimesTen release. If JMS/XLA finds other versions of the GDK loaded in the JVM, it displays a severe warning and continues processing. You can find out the GDK version supported by JMS/XLA by entering the following commands:

```
$ cd install_dir/lib
$ java -cp ./orai18n.jar oracle.i18n.util.GDKOracleMetaData -version
```

Also see "Compiling Java applications" on page 1-2.

Connecting to XLA

To connect to XLA so you can receive updates, use a JMS connection factory to create a connection. Then use the connection to establish a session. When you are ready to start processing updates, call `start()` on the connection to enable message dispatching. This is shown in Example 3–5 that follows, from the `syncJMS` Quick Start demo.

Example 3–5 Connecting to XLA

```
/** JMS connection */
private javax.jms.TopicConnection connection;
/** JMS session */
private TopicSession session;
...
// get Connection
Context messaging = new InitialContext();
TopicConnectionFactory connectionFactory =
        (TopicConnectionFactory)messaging.lookup("TopicConnectionFactory");
connection = connectionFactory.createTopicConnection();
connection.start();
...
// get Session
session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
```

Monitoring tables for updates

Before you can start receiving updates, you must specify to XLA which tables you want to monitor for changes.

To subscribe to changes and turn on XLA publishing for a table, call the `ttXlaSubscribe` built-in procedure through JDBC.

When you use `ttXlaSubscribe` to enable XLA publishing for a table, you must specify parameters for the name of the table and the name of the bookmark that will be used to track the table:

```
ttXlaSubscribe(user.table, mybookmark)
```

For example, call `ttXlaSubscribe` by the JDBC `CallableStatement` interface:

```
Connection con;

CallableStatement cStmt;
...
cStmt = con.prepareCall("{call ttXlaSubscribe(user.table, mybookmark)}");
cStmt.execute();
```

Use `ttXlaUnsubscribe` to unsubscribe from the table during shutdown. For more information, see "Unsubscribing from a table" on page 3-13.

The application can verify table subscriptions by checking the `SYS.XLASUBSCRIPTIONS` system table.

For more information about using TimesTen built-in procedures in a Java application, see "Using CALL to execute procedures and functions" on page 2-24.

Receiving and processing updates

You can receive XLA updates either synchronously or asynchronously.

To receive and process updates for a topic synchronously, perform the following tasks.

1. Create a durable `TopicSubscriber` instance to subscribe to a topic.
2. Call `receive()` or `receiveNoWait()` on your subscriber to get the next available update.
3. Process the returned `MapMessage` instance.

To receive and process updates for a topic asynchronously, perform the following tasks.

1. Create a `MessageListener` instance to process the updates.
2. Create a durable `TopicSubscriber` instance to subscribe to a topic.
3. Register the `MessageListener` with the `TopicSubscriber`.
4. Start the connection.

> **Note:** You may miss messages if you do not register the `MessageListener` before you start the connection. If the connection is already started, stop the connection, register the `MessageListener`, then start the connection.

5. Wait for messages to arrive. You can call the `Object` method `wait()` to wait for messages if your application does not have to do anything else in its main thread.

When an update is published, the `MessageListener` method `onMessage()` is called and the message is passed in as a `MapMessage` instance.

The application can verify table subscriptions by checking the `SYS.XLASUBSCRIPTIONS` system table.

Example 3–6, from the `asyncJMS` Quick Start demo, uses a listener to process updates asynchronously.

Example 3–6 Using a listener to process updates asynchronously

```
MyListener myListener = new MyListener(outStream);

outStream.println("Creating consumer for topic " + topic);
Topic xlaTopic = session.createTopic(topic);
bookmark = "bookmark";
TopicSubscriber subscriber = session.createDurableSubscriber(xlaTopic, bookmark);

// After setMessageListener() has been called, myListener's onMessage
// method will be called for each message received.
subscriber.setMessageListener(myListener);
```

Note that `bookmark` must already exist. You can use JDBC and the `ttXlaBookmarkCreate` built-in procedure to create a bookmark. Also, the `TopicSubscriber` must be a durable subscriber. XLA connections are designed to be durable. XLA bookmarks make it possible to disconnect from a topic and then reconnect to start receiving updates where you left off. The string you pass in as the subscription identifier when you create a durable subscriber is used as the XLA bookmark name.

You can call `unsubscribe()` on the JMS `TopicSession` to delete the XLA bookmark used by the subscriber when the application shuts down. This causes a new bookmark to be created when the application is restarted.

When you receive an update, you can use the `MapMessage` getter methods to extract information from the message and then perform whatever processing your application requires. The TimesTen `XlaConstants` class defines constants for the update types and special message fields for use in processing XLA update messages.

The first step is typically to determine what type of update the message contains. You can use the `MapMessage` method `getInt()` to get the contents of the `__TYPE` field,

and compare the value against the numeric constants defined in the `XlaConstants` class.

In Example 3–7, from the `asyncJMS` Quick Start demo, the method `onMessage()` extracts the update type from the `MapMessage` object and displays the action that the update signifies.

Example 3–7 Determining the update type

```
public void onMessage(Message message)
{
  MapMessage mapMessage = (MapMessage)message;
  String messageType = null;
  /** Standard output stream */
  private static PrintStream outStream = System.out;

  if (message == null)
  {
    errStream.println("MyListener: update message is null");
    return ;
  }

  try
  {
    outStream.println();
    outStream.println("onMessage: got a " + mapMessage.getJMSType() + " message");

    // Get the type of event (insert, update, delete, drop table, etc.).
    int type = mapMessage.getInt(XlaConstants.TYPE_FIELD);
    if (type == XlaConstants.INSERT)
    {
      outStream.println("A row was inserted.");
    }
    else if (type == XlaConstants.UPDATE)
    {
      outStream.println("A row was updated.");
    }
    else if (type == XlaConstants.DELETE)
    {
      outStream.println("A row was deleted.");
    }
    else
    {

      // Messages are also received for DDL events such as CREATE TABLE.
      // This program processes INSERT, UPDATE, and DELETE events,
      // and ignores the DDL events.
      return ;
    }
  ...
  }
...
}
```

When you know what type of message you have received, you can process the message according to the application's needs. To get a list of all of the fields in a message, you can call the `MapMessage` method `getMapNames()`. You can retrieve individual fields from the message by name.

Example 3–8, from the `asyncJMS` Quick Start demo, extracts the column values from insert, update, and delete messages using the column names.

Example 3–8 Extracting column values

```
/** Standard output stream */
private static PrintStream outStream = System.out;
...
if (type == XlaConstants.INSERT
 || type == XlaConstants.UPDATE
 || type == XlaConstants.DELETE)
{

  // Get the column values from the message.
  int cust_num = mapMessage.getInt("cust_num");
  String region = mapMessage.getString("region");
  String name = mapMessage.getString("name");
  String address = mapMessage.getString("address");

  outStream.println("New Column Values:");
  outStream.println("cust_num=" + cust_num);
  outStream.println("region=" + region);
  outStream.println("name=" + name);
  outStream.println("address=" + address);
}
```

For detailed information about the contents of XLA update messages, see "JMS/XLA MapMessage contents" on page 6-1. For information about how TimesTen column types map to JMS data types and the getter methods used to retrieve the column values, see "Data type support" on page 6-10.

Terminating a JMS/XLA application

When the XLA application has finished reading from the transaction log, it should gracefully exit by closing the XLA connection, deleting any unneeded bookmarks, and unsubscribing from any tables to which you explicitly subscribed.

Closing the connection

To close the connection to XLA, call `close()` on the `Connection` object.

After a connection has been closed, any attempt to use it, its sessions, or its subscribers will throw an `IllegalStateException`. You can continue to use messages received through the connection, but you cannot call the `acknowledge()` method on the received message after the connection is closed.

Deleting bookmarks

Deleting XLA bookmarks during shutdown is optional. Deleting a bookmark enables the disk space associated with any unread update records in the transaction log to be freed.

If you do not delete the bookmark, it can be reused by a durable subscriber. If the bookmark is available when a durable subscriber reconnects, the subscriber will receive all unacknowledged updates published since the previous connection was terminated. Keep in mind that when a bookmark exists with no application reading from it, the transaction log will continue to grow and the amount of disk space consumed by your database will increase.

To delete a bookmark, you can simply call `unsubscribe` on the JMS Session, which invokes the `ttXlaBookmarkDelete` built-in procedure to remove the XLA bookmark.

> **Note:** You cannot delete replicated bookmarks while the replication agent is running.

Unsubscribing from a table

To turn off XLA publishing for a table, use the `ttXlaUnsubscribe` built-in procedure. If you use `ttXlaSubscribe` to enable XLA publishing for a table, you should use `ttXlaUnsubscribe` to unsubscribe from the table when shutting down your application.

> **Note:** If you want to drop a table, you must unsubscribe from it first.

When you unsubscribe from a table, specify the name of the table and the name of the bookmark used to track the table:

```
ttXlaUnsubscribe(user.table, mybookmark)
```

The following example calls `ttXlaUnSubscribe` through a `CallableStatement` object.

Example 3–9 Unsubscribing from a table

```
Connection con;

CallableStatement cStmt;
...
cStmt = con.prepareCall("{call ttXlaUnSubscribe(user.table, mybookmark)}");
cStmt.execute();
```

For more information about using TimesTen built-in procedures in a Java application, see "Using CALL to execute procedures and functions" on page 2-24.

Using JMS/XLA as a replication mechanism

If the TimesTen replication solutions described in *Oracle TimesTen In-Memory Database TimesTen to TimesTen Replication Guide* do not meet your needs, you can use JMS/XLA to replicate updates from a source database to a target database.

Applying JMS/XLA messages to a target database

The source database generates JMS/XLA messages. To apply the messages to a target database, you must extract the XLA descriptor from them. Use the `MapMessage` interface to extract the update descriptor:

```
MapMessage message;
/**
 *...other code
 */
try {
  byte[]updateMessage=
    mapMessage.getBytes(XlaConstants.UPDATE_DESCRIPTOR_FIELD);
}
```

```
catch (JMSException jex){
/**
 *...other code
 */
}
```

The target database may reside on a different system from the source database. The update descriptor is returned as a byte array and can be serialized for network transmission.

You must create a target database object that represents the target database so you can apply the objects from the source database. You can create a target database object named `myTargetDataStore` as an instance of the `TargetDataStoreImpl` class. For example:

```
TargetDataStore myTargetDataStore=
   new TargetDataStoreImpl("DSN=sampleDSN");
```

Apply messages to `myTargetDataStore` by using the `TargetDataStore` method `apply()`. For example:

```
myTargetDataStore.apply(updateDescriptor);
```

By default, TimesTen checks for conflicts on the target database before applying the update. If the target database has information that is later than the update, `TargetDataStore` throws an exception. If you do not want TimesTen to check for conflicts, use the `TargetDataStore` method `setUpdateConflictCheckFlag()` to change the behavior.

By default, TimesTen commits the update to the database based on commit flags and transaction boundaries contained in the update descriptor. If you want the application to perform manual commits instead, use the `setAutoCommitFlag()` method to change the autocommit flag. To perform a manual commit on `myTargetDataStore`, use the following command:

```
myTargetDataStore.commit();
```

You can perform a rollback if errors occur during the application of the update. Use the following command for `myTargetDataStore`:

```
myTargetDataStore.rollback();
```

Close `myTargetDataStore` by using the following command:

```
myTargetDataStore.close;
```

See "JMS/XLA replication API" on page 6-13 for more information about the `TargetDataStore` interface.

TargetDataStore error recovery

Invoking `TargetDataStore` can yield transient and permanent errors.

`TargetDataStore` methods return a nonzero value when transient errors occur. The application can retry the operation and is responsible for monitoring update descriptors that must be reapplied. For more information about transient XLA errors, see "Handling XLA errors" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

`TargetDataStore` methods return a `JMSException` object for permanent errors. If the application receives a permanent error, it should verify that the database is valid. If

the database is invalid, the target database object should be closed and a new one should be created. Other types of permanent errors may require manual intervention.

The following example shows how to recover errors from a `TargetDataStore` object.

Example 3–10 Recovering errors

```
TargetDataStore theTargetDataStore;
byte[] updateDescriptor;
int rc;

// Other code
try {
  ...
  if ( (rc = theTargetDataStore.apply(updateDescriptor) ) == 0 ) {
    // apply successful
  }
  else {
    // Transient error. Retry later.
  }
 }
catch (JMSException jex) {
  if (theTargetDataStore.isDataStoreValid() ) {
    // Database valid; permanent error that may need Administrator intervention.
  }
  else {
    try {
      theTargetDataStore.close();
  }
  catch (JMSException closeEx) {
    // Close errors are not usual. This may need Administrator intervention.
  }
}
```

4

Distributed Transaction Processing: JTA

This chapter describes the TimesTen implementation of the Java Transaction API (JTA).

The TimesTen implementation of the Java JTA interfaces is intended to enable Java applications, application servers, and transaction managers to use TimesTen resource managers in distributed transaction processing (DTP) environments. The TimesTen implementation is supported for use by the Oracle WebLogic Server.

The purpose of this chapter is to provide information specific to the TimesTen implementation of JTA and is intended to be used with the following documents:

- The JTA and JDBC documentation available from the following locations:

  `http://www.oracle.com/technetwork/java/javaee/tech/`

  `http://www.oracle.com/technetwork/java/javase/tech/`

- WebLogic documentation, available through the following location:

  `http://www.oracle.com/technetwork/middleware/weblogic/documentation`

As TimesTen JTA is built on top of the TimesTen implementation of the X/Open XA standard, much of the discussion here is in terms of underlying XA features. You can also refer to "Distributed Transaction Processing: XA" in *Oracle TimesTen In-Memory Database C Developer's Guide*.

This chapter includes the following topics:

Important:

- The TimesTen XA implementation does not work with the Oracle In-Memory Database Cache (IMDB Cache). The start of any XA transaction will fail if the cache agent is running.
- You cannot execute an XA transaction if replication is enabled.
- Do not execute DDL statements within an XA transaction.

Overview of JTA

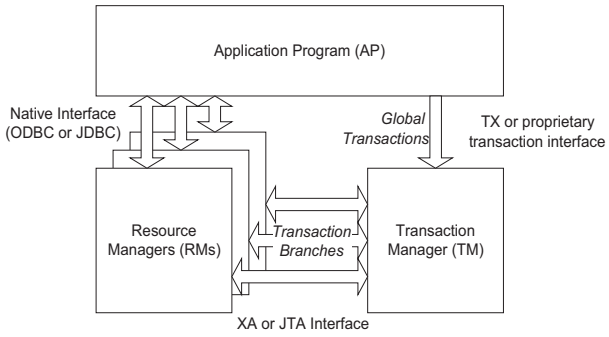
This section provides a brief overview of the following XA concepts.

- Two-phase commit

X/Open DTP model

Figure 4–1 illustrates the interfaces defined by the X/Open DTP model.

Figure 4–1 Distributed transaction processing model



The TX interface is what applications use to communicate with a transaction manager. The figure shows an application communicating global transactions to the transaction manager. In the DTP model, the transaction manager breaks each global transaction down into multiple branches and distributes them to separate resource managers for service. It uses the JTA interface to coordinate each transaction branch with the appropriate resource manager.

In the context of TimesTen JTA, the resource managers can be a collection of TimesTen databases, or databases in combination with other commercial databases that support JTA.

Global transaction control provided by the TX and JTA interfaces is distinct from local transaction control provided by the native JDBC interface. It is generally best to maintain separate connections for local and global transactions. Applications can obtain a connection handle to a TimesTen resource manager to initiate both local and global transactions over the same connection.

Two-phase commit

In a JTA implementation, the transaction manager commits the distributed branches of a global transaction by using a two-phase commit protocol.

1. In phase 1, the transaction manager directs each resource manager to prepare to commit, which is to verify and guarantee it can commit its respective branch of the global transaction. If a resource manager cannot commit its branch, the transaction manager rolls back the entire transaction in phase 2.
2. In phase 2, the transaction manager either directs each resource manager to commit its branch or, if a resource manager reported it was unable to commit in phase 1, rolls back the global transaction.

Note the following optimizations.

- If a global transaction is determined by the transaction manager to have involved only one branch, it skips phase 1 and commits the transaction in phase 2.
- If a global transaction branch is read-only, where it does not generate any transaction log records, the transaction manager commits the branch in phase 1 and skips phase 2 for that branch.

> **Note:** The transaction manager considers the global transaction committed if and only if all branches successfully commit.

Using JTA in TimesTen

This section discusses the following considerations for using JTA in TimesTen:

- TimesTen database requirements for XA
- Global transaction recovery in TimesTen
- XA error handling in TimesTen

TimesTen database requirements for XA

To guarantee global transaction consistency, TimesTen XA transaction branches must be durable. The TimesTen implementation of the `xa_prepare()`, `xa_rollback()`, and `xa_commit()` functions log their actions to disk, regardless of the value set in the `DurableCommits` general connection attribute or by the `ttDurableCommit` built-in procedure. If you must recover from a failure, both the resource manager and the TimesTen transaction manager have a consistent view of which transaction branches were active in a prepared state at the time of failure.

Rollback of transactions requires transaction logging, which is always enabled with XA.

Global transaction recovery in TimesTen

When a database is loaded from disk to recover after a failure or unexpected termination, any global transactions that were prepared but not committed are left pending, or in doubt. Normal processing is not enabled until the disposition of all in-doubt transactions has been resolved.

After connection and recovery are complete, TimesTen checks for in-doubt transactions. If there are no in-doubt transactions, operation proceeds as normal. If there are in-doubt transactions, other connections may be created, but virtually all operations are prohibited on those connections until the in-doubt transactions are resolved. Any other JDBC calls result in the following error:

```
Error 11035 - "In-doubt transactions awaiting resolution in recovery must be
resolved first"
```

The list of in-doubt transactions can be retrieved through the XA implementation of `xa_recover()`, then dealt with through the XA call `xa_commit()`, `xa_rollback()`, or `xa_forget()`, as appropriate. After all the in-doubt transactions are cleared, operations proceed normally.

This scheme should be adequate for systems that operate strictly under control of the transaction manager, since the first thing the transaction manager should do after connect is to call `xa_recover()`.

If the transaction manager is unavailable or cannot resolve an in-doubt transaction, you can use the `ttXactAdmin` utility to independently commit or abort the individual transaction branches. Be aware, however, that these `ttXactAdmin` options require `ADMIN` privilege. See "ttXactAdmin" in *Oracle TimesTen In-Memory Database Reference*.

XA error handling in TimesTen

The XA specification has a limited, strictly defined set of errors that can be returned from XA interface calls. The ODBC `SQLError` mechanism returns XA defined errors, along with any additional information.

The TimesTen XA related errors begin at number 11000. Errors 11002 through 11020 correspond to the errors defined by the XA standard.

See "Warnings and Errors" in *Oracle TimesTen In-Memory Database Error Messages and SNMP Traps* for the complete list of errors.

Using the JTA API

The TimesTen implementation of JTA provides an API consistent with that specified in the JTA specification. TimesTen JTA operates on JDK 1.4 and above.

This section covers the following topics for using the JTA API:

- Required packages
- Creating a TimesTen XAConnection object
- Creating XAResource and Connection objects

Regarding how to register a TimesTen DSN with WebLogic, information on configuring TimesTen for application servers and object-relational mapping frameworks is available in the TimesTen Quick Start. Click **J2EE and OR Mapping** under Configuration and Setup.

Required packages

The TimesTen JDBC and XA implementations are available in the following packages:

```
com.timesten.jdbc.*;
com.timesten.jdbc.xa.*;
```

Your application should also import these standard packages:

```
import java.sql.*;
import javax.sql.*;
import javax.transaction.xa.*;
```

Creating a TimesTen XAConnection object

Connections to XA data sources are established through `XADataSource` objects. You can create an `XAConnection` object for your database by using the `TimesTenXADataSource` instance as a connection factory. `TimesTenXADataSource` implements the `javax.sql.XADataSource` interface.

After creating a new `TimesTenXADataSource` instance, use the `setUrl()` method to specify a database. The URL should look similar to the following.

- For a direct connection:

  ```
  jdbc:timesten:direct:DSNname
  ```

- For a client connection:

```
jdbc:timesten:client:DSNname
```

You can also optionally use the `setUser()` and `setPassword()` methods to set the ID and password for a specific user.

> **Note:** Privilege must be granted to connect to a database. Refer to "Access control for connections" on page 2-8.

Example 4–1 Creating a TimesTen XA data source object

In this example, the `TimesTenXADataSource` object is used as a factory to create a new TimesTen XA data source object. Then the URL that identifies the TimesTen DSN (`dsn1`), the user name (`myName`), and the password (`myPasswd`) are set for this `TimesTenXADataSource` instance. Then the `getXAConnection()` method is used to return a connection to the object, `xaConn`.

```
TimesTenXADataSource xads = new TimesTenXADataSource();

xads.setUrl("jdbc:timesten:direct:dsn1");
xads.setUser("myName");
xads.setPassword("myPassword");

XAConnection xaConn = null;
try {
    xaConn = xads.getXAConnection();
}
catch (SQLException e){
    e.printStackTrace();
    return;
}
```

You can create multiple connections to an XA data source object. This example creates a second connection, `xaConn2`:

```
XAConnection xaConn  = null;
XAConnection xaConn2 = null;
try {
    xaConn  = xads.getXAConnection();
    xaConn2 = xads.getXAConnection();
}
```

Example 4–2 Creating multiple TimesTen XA data source objects

This example creates two instances of `TimesTenXADataSource` for the databases named `dsn1` and `dsn2`. It then creates a connection for `dsn1` and two connections for `dsn2`.

```
TimesTenXADataSource xads = new TimesTenXADataSource();

xads.setUrl("jdbc:timesten:direct:dsn1");
xads.setUser("myName");
xads.setPassword("myPassword");

XAConnection xaConn1 = null;
XAConnection xaConn2 = null;
XAConnection xaConn3 = null;

try {
```

```
   xaConn1 = xads.getXAConnection(); // connect to dsn1
}
catch (SQLException e){
   e.printStackTrace();
   return;
}

xads.setUrl("jdbc:timesten:direct:dsn2");
xads.setUser("myName");
xads.setPassword("myPassword");

try {
   xaConn2 = xads.getXAConnection(); // connect to dsn2
   xaConn3 = xads.getXAConnection(); // connect to dsn2
}
catch (SQLException e){
   e.printStackTrace();
   return;
}
```

Note: Once an `XAConnection` is established, autocommit is off.

Creating XAResource and Connection objects

After using `getXAConnection()` to obtain an `XAConnection` object, you can use the `XAConnection` method `getXAResource()` to obtain an `XAResource` object, then the `XAConnection` method `getConnection()` to obtain a `Connection` object for the underlying connection.

Example 4–3 Getting an XA resource object and a connection

```
//get an XAResource
XAResource xaRes = null;
try {
   xaRes = xaConn.getXAResource();
}catch (SQLException e){
   e.printStackTrace();
   return;
}

//get an underlying physical Connection
Connection conn = null;
try {
   conn = xaConn.getConnection();
}catch (SQLException e){
   e.printStackTrace();
   return;
}
```

From this point, you can use the same connection, `conn`, for both local and global transactions. Be aware of the following, however.

- You must commit or roll back an active local transaction before starting a global transaction. Otherwise you will get the `XAException` exception `XAER_OUTSIDE`.
- You must end an active global transaction before initiating a local transaction, otherwise you will get a `SQLException`, "Illegal combination of local transaction and global (XA) transaction."

5

Application Tuning

This chapter provides tips on how to tune a Java application to run optimally on a TimesTen database. See "TimesTen Database Performance Tuning" in *Oracle TimesTen In-Memory Database Operations Guide* for more general tuning tips.

This chapter is organized as follows:

- Tuning Java applications
- Tuning JMS/XLA applications

Tuning Java applications

This section describes general principles to consider when tuning Java applications for TimesTen. It includes the following topics:

- Use prepared statement pooling
- Use arrays of parameters for batch execution
- Bulk fetch rows of TimesTen data
- Use the ResultSet method getString() sparingly
- Avoid data type conversions

> **Note:** Also see "Working with TimesTen result sets: hints and restrictions" on page 2-10 and the notes in "Binding parameters and executing statements" on page 2-11.

Use prepared statement pooling

TimesTen supports prepared statement pooling for pooled connections, as discussed in the JDBC 3.0 specification, through the TimesTen `ObservableConnectionDS` class. This is the TimesTen implementation of `ConnectionPoolDataSource`. Note that statement pooling is transparent to an application. Use of the `PreparedStatement` object, including preparing and closing the statement, is no different.

Enable prepared statement pooling and specify the maximum number of statements in the pool by calling the `ObservableConnectionDS` method `setMaxStatements()`. A value of 0, the default, disables prepared statement pooling. Any integer value greater than 0 enables prepared statement pooling with the value taken as the maximum number of statements. Once set, this value should not be changed.

Prepared statements or callable statements will be pooled at the time of creation if the pool has not reached its capacity. In Java 6, you can remove a prepared statement or

callable statement from the pool by calling `setPoolable(false)` on the statement object. After the statement is closed, it will be removed from the pool.

Important: With prepared statement pooling, JDBC considers two statements to be identical if their SQL (including comments) is identical, regardless of other considerations such as optimizer settings. Do not use prepared statement pooling in a scenario where different optimizer hints may be applied to statements that are otherwise identical. In this scenario, a statement execution may result in the use of an identical statement from the pool with an unanticipated optimizer setting.

Use arrays of parameters for batch execution

You can improve performance by using groups, referred to as *batches*, of statement executions, calling the `addBatch()` and `executeBatch()` methods for `Statement` or `PreparedStatement` objects.

A batch can consist of a set of `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statements. Statements that return result sets, such as `SELECT` statements, are not allowed in a batch. A SQL statement is added to a batch by calling `addBatch()` on the statement object. The set of SQL statements associated with a batch are executed through the `executeBatch()` method.

For `PreparedStatement` objects, a batch consists of repeated executions of a statement using different input parameter values. For each set of input values, create the batch by using appropriate `set`*`XXX`*`()` calls followed by the `addBatch()` call. The batch is executed by the `executeBatch()` method.

TimesTen recommends the following batch sizes for Release 11.2.1:

- 256 for `INSERT` statements
- 31 for `UPDATE` statements
- 31 for `DELETE` statements
- 31 for `MERGE` statements

Example 5–1 Batching statements

```
// turn off autocommit
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");

// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
conn.commit ();
```

Example 5–2 Batching prepared statements

```
// turn off autocommit
conn.setAutoCommit(false);
// prepare the statement
PreparedStatement stmt = conn.prepareStatement
                         ("INSERT INTO employees VALUES (?, ?)");
```

```
// first set of parameters
stmt.setInt(1, 2000);
stmt.setString(2, "Kelly Kaufmann");
stmt.addBatch();

// second set of parameters
stmt.setInt(1, 3000);
stmt.setString(2, "Bill Barnes");
stmt.addBatch();

// submit the batch for execution. Check update counts
int[] updateCounts = stmt.executeBatch();
conn.commit ();
```

For either a `Statement` or `PreparedStatement` object, the `executeBatch()` method returns an array of update counts (`updateCounts[]` in Example 5–1 and Example 5–2 above), with one element in the array for each statement execution. The value of each element can be any of the following:

- A number indicating how many rows in the database were affected by the corresponding statement execution.
- `SUCCESS_NO_INFO`, indicating the corresponding statement execution was successful, but the number of affected rows is unknown.
- `EXECUTE_FAILED`, indicating the corresponding statement execution failed.

  Once there is a statement execution with `EXECUTE_FAILED` status, no further statement executions will be attempted.

For more information about using the JDBC batch update facility, refer to the Javadoc for the `Statement` interface, particularly information about the `executeBatch()` method, at the following location:

`http://download.oracle.com/javase/1.5.0/docs/api/`

Bulk fetch rows of TimesTen data

TimesTen provides an extension that allows an application to fetch multiple rows of data. For applications that retrieve large amounts of TimesTen data, fetching multiple rows can increase performance greatly. However, when using Read Committed isolation level, locks are held on all rows being retrieved until the application has received all the data, decreasing concurrency. For more information on this feature, see "Fetching multiple rows of data" on page 2-10.

Use the ResultSet method getString() sparingly

Because Java strings are immutable, the `ResultSet` method `getString()` must allocate space for a new string in addition to translating the underlying C string to a Unicode string, making it a costly call.

In addition, you should not call `getString()` on primitive numeric types, like `byte` or `int`, unless it is absolutely necessary. It is much faster to call `getInt()` on an integer column, for example.

Avoid data type conversions

TimesTen instruction paths are so short that even small delays due to data conversion can cause a relatively large percentage increase in transaction time.

Use the appropriate `getXXX()` method on a `ResultSet` object for the data type of the data in the underlying database. For example, if the data type of the data is `DOUBLE`, to avoid data conversion in the JDBC driver you should call `getDouble()`. Similarly, use the appropriate `setXXX()` method on the `PreparedStatement` object for the input parameter in an SQL statement. For example, if you are inserting data into a `CHAR` column using a `PreparedStatement`, you should use `setString()`.

Tuning JMS/XLA applications

This section contains specific performance tuning tips for applications that use the JMS/XLA API. JMS/XLA has some overhead that makes it slower than using the C XLA API. In the C API, records are returned to the user in a batch. In the JMS model an object is instantiated and each record is presented one at a time in a callback to the `MessageListener` method `onMessage()`. High performance applications can use some tuning to overcome some of this overhead.

This section includes the following topics:

- Configure xlaPrefetch parameter
- Reduce frequency of calls to ttXlaAcknowledge
- Handling high event rates

Note: See "Access control impact on XLA" on page 3-8 for access control considerations relevant to JMS/XLA.

Configure xlaPrefetch parameter

The code underlying the JMS layer that reads the transaction log is more efficient if it can fetch as many rows as possible before presenting the object/rows to the user. The amount of prefetching is controlled in the `jmsxla.xml` configuration file with the `xlaPrefetch` parameter. Set the prefetch count to a large value like 100 or 1000.

Reduce frequency of calls to ttXlaAcknowledge

Calls to the C XLA function `ttXlaAcknowledge` move the bookmark and involve updates to system tables, so one way to increase throughput is to wait until several transactions have been detected before issuing the call. The reader application must have some tolerance for seeing the same set of records more than once. Moving the bookmark can be done manually using the `Session` object `CLIENT_ACKNOWLEDGE` mode when instantiating a session:

```
Session session = connection.createSession (false, Session.CLIENT_ACKNOWLEDGE);
```

For many applications, setting this value to 100 is a reasonable choice.

Refer to "ttXlaAcknowledge" in *Oracle TimesTen In-Memory Database C Developer's Guide* for information about this function.

Handling high event rates

The synchronous interface is suitable only for applications with low event rates and for which `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE` acknowledgment modes are acceptable. Applications that require `CLIENT_ACKNOWLEDGE` acknowledgment mode and applications with high event rates should use the asynchronous interface for receiving updates. They should acknowledge the messages on the callback thread

itself if they are using `CLIENT_ACKNOWLEDGEMENT` as acknowledgment mode. See "Receiving and processing updates" on page 3-9.

6

JMS/XLA Reference

This chapter provides reference information for the JMS/XLA API. It includes the following topics:

- JMS/XLA MapMessage contents
- DML event data formats
- DDL event data formats
- Data type support
- JMS classes for event handling
- JMS/XLA replication API
- JMS message header fields

> **Note:** "Access control impact on XLA" on page 3-8 introduces the effects of TimesTen access control features on XLA functionality.

JMS/XLA MapMessage contents

A `javax.jms.MapMessage` contains a set of typed name and value pairs that correspond to the fields in an XLA update header, which is published as the C structure `ttXlaUpdateDesc_t`. The fields contained in a `MapMessage` instance depend on what type of update it is.

XLA update types

Each `MapMessage` returned by the JMS/XLA API contains at least one name and value pair, `__TYPE` (with 2 underscores), that identifies the type of update described in the message as an integer value. The types are specified as integer values. As a convenience, you can use the constants defined in `com.timesten.dataserver.jmsxla.XlaConstants` to compare against the integer types. Table 6–1 shows the supported types.

Table 6–1 XLA update types

Type	Description
`ADD_COLUMNS`	Indicates that columns have been added.
`COMMIT_FIELD`	The name of the field in a message that contains a commit.
`COMMIT_ONLY`	Indicates that a commit has occurred.

Table 6–1 (Cont.) XLA update types

Type	Description
CONTEXT_FIELD	The name of the field in a message that contains the context value passed to the ttApplicationContext procedure as a byte array.
CREATE_INDEX	Indicates that an index has been created.
CREATE_SEQ	Indicates that a sequence has been created.
CREATE_SYNONYM	Indicates that a synonym has been created.
CREATE_TABLE	Indicates that a table has been created.
CREATE_VIEW	Indicates that a view has been created.
DELETE	Indicates that a row has been deleted.
DROP_COLUMNS	Indicates that columns have been dropped.
DROP_INDEX	Indicates that an index has been dropped.
DROP_SEQ	Indicates that a sequence has been dropped.
DROP_SYNONYM	Indicates that a synonym has been dropped.
DROP_TABLE	Indicates that a table has been dropped.
DROP_VIEW	Indicates that a view has been dropped.
FIRST_FIELD	The name of the field that contains the flag that indicates the first record in a transaction.
INSERT	Indicates that a row has been inserted.
MTYP_FIELD	The name of the field in a message that contains type information.
MVER_FIELD	The name of the field in a message that contains the transaction log file number of the XLA record.
NULLS_FIELD	The name of the field in a message that contains the list of fields that have null values.
REPL_FIELD	The name of the field in a message that contains the flag that indicates that the update was applied by replication.
TBLNAME_FIELD	The name of the field in a message that contains the table name.
TBLOWNER_FIELD	The name of the field in a message that specifies the table owner.
TRUNCATE	Indicates that a table has been truncated.
TYPE_FIELD	The name of the field in a message that specifies the message type.
UPDATE	Indicates that a row has been updated.
UPDATE_DESCRIPTOR_FIELD	The name of the field that returns a ttXlaUpdateDesc_t structure as a byte array.
UPDATED_COLUMNS_FIELD	The name of the field in a message that contains the list of updated columns.

XLA flags

For all update types, the MapMessage contains name and value pairs that indicate the following.

- Whether this is the first record of a transaction
- Whether this is the last record of a transaction
- Whether the update was performed by replication
- Which table was updated
- The owner of the updated table

The name and value pairs that contain these XLA flags are described in Table 6–2. Each name is preceded by two underscores.

Table 6–2 JMS/XLA flags

Name	Description	Corresponding ttXlaUpdateDesc_t flag
__AGING_DELETE	Indicates that a delete was due to aging. The flag is present only if the XLA update record is due to an aging delete. The XlaConstants constant AGING_DELETE_FIELD represents this flag.	TT_AGING
__CASCADING_DELETE	Indicates that a delete was due to a cascading delete. The flag is present only if the XLA update record is due to a cascading delete. The XlaConstants constant CASCADING_DELETE_FIELD represents this flag.	TT_CASCDEL
__COMMIT	Indicates that this is the last record in a transaction and that a commit was performed after this operation. Only included in the MapMessage if TT_UPDCOMMIT is on. The XlaConstants constant COMMIT_FIELD represents this flag.	TT_UPDCOMMIT
__FIRST	Indicates that this is the first record in a new transaction. Only included in the MapMessage if TT_UPDFIRST is on. The XlaConstants constant FIRST_FIELD represents this flag.	TT_UPDFIRST
__REPL	Indicates that this change was applied to the database through replication. Only included in the MapMessage if TT_UPDREPL is on. The XlaConstants constant REPL_FIELD represents this flag.	TT_UPDREPL
__UPDCOLS	Only used for UPDATETUP records, this flag indicates that the XLA update descriptor contains a list of columns that were actually modified by the operation. Specified as a string that contains a semicolon-delimited list of column names. Only included in the MapMessage if TT_UPDCOLS is on. The XlaConstants constant UPDATE_COLUMNS_FIELD represents this flag.	TT_UPDCOLS

Note: The `XlaConstants` interface is in the `com.timesten.dataserver.jmsxla` package.

Applications can use the `MapMessage` method `itemExists()` to determine whether a flag is present, and `getBoolean()` to determine whether a flag is set. As input, specify the `XlaConstants` constant that corresponds to the flag, such as `XlaConstants.AGING_DELETE_FIELD`.

Example 6–1 Check for commit

Equivalent to using `TT_UPDCOMMIT` in XLA, you can use the following test in JMS/XLA to see whether this is the last record in a transaction and that a commit was performed after the operation.

```
if (MapMessage.getBoolean(XlaConstants.COMMIT_FIELD) ) { // Field is set
   ...
}
```

DML event data formats

Many DML operations generate XLA updates that can be monitored by XLA event handlers. This section describes the contents of the `MapMessage` objects that are generated for these operations.

Table data

For `INSERT`, `UPDATE` and `DELETE` operations, `MapMessage` contains two name and value pairs, `__TBLOWNER` and `__TBLNAME`. These fields describe the name and owner of the table that is being updated. For example, for a table `SCOTT.EMPLOYEES`, any related `MapMessage` contains a field `__TBLOWNER` with the string value "SCOTT" and a field `__TBLNAME` with the string value "EMPLOYEES".

Row data

For `INSERT` and `DELETE` operations, a complete image of the inserted or deleted row is included in the message and all column values are available.

For `UPDATE` operations, the complete "before" and "after" images of the row are available, along with a list of column numbers indicating which columns were modified. Access the column values using the names of the columns. The column names in the "before" image all begin with a single underscore. For example, `columnname` contains the new value and `_columnname` contains the old value.

If the value of a column is `NULL`, it is omitted from the column list. The `__NULLS` name and value pair contains a semicolon-delimited list of the columns that contain `NULL` values.

Context information

If the `ttApplicationContext` built-in procedure was used to encode context information in an XLA record, that information is included in the `__CONTEXT` name and value pair in the `MapMessage`. If no context information is provided, the `__CONTEXT` value is not included in the `MapMessage`.

DDL event data formats

Many data definition language (DDL) operations generate XLA updates that can be monitored by XLA event handlers. This section describes the contents of the MapMessage objects that are generated for these operations.

CREATE_TABLE

Messages with __TYPE=1 (XlaConstants.CREATE_TABLE) indicate that a table has been created. Table 6–3 shows the name and value pairs that are included in a MapMessage generated for a CREATE_TABLE operation.

Table 6–3 CREATE_TABLE data provided in update messages

Name	Value
OWNER	String value of the owner of the created table.
NAME	String value of the name of the created table.
PK_COLUMNS	String value containing the names of the columns in the primary key for this table. If the table has no primary key, the PK_COLUMNS value is not specified. Format: <col1name>[;<col2name> [;<col3name>[;...]]]
COLUMNS	String value containing the names of the columns in the table. Format: <col1name>[;<col2name> [;<col3name>[;...]]] **Note:** For each column in the table, additional name and value pairs that describe the column are included in MapMessage.
*_column_name*_TYPE	Integer value representing the data type of this column. From java.sql.Types.
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC or DECIMAL).
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC or DECIMAL).
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR, VARCHAR, BINARY, or VARBINARY).
*_column_name*_NULLABLE	Boolean value indicating whether this column can have a NULL value.
*_column_name*_OUTOFLINE	Boolean value indicating whether this column is stored in the inline or out-of-line part of the tuple.
*_column_name*_INPRIMARYKEY	Boolean value indicating whether this column is part of the primary key of the table.

DROP_TABLE

Messages with __TYPE=2 (XlaConstants.DROP_TABLE) indicate that a table has been dropped. Table 6–4 shows the name and value pairs that are included in a MapMessage generated for a DROP_TABLE operation.

Table 6–4 DROP_TABLE data provided in update messages

Name	Value
OWNER	String value of the owner of the sequence.
NAME	String value of the name of the dropped sequence.

CREATE_INDEX

Messages with __TYPE=3 (XlaConstants.CREATE_INDEX) indicate that an index has been created. Table 6–5 shows the name and value pairs that are included in a MapMessage generated for a CREATE_INDEX operation.

Table 6–5 CREATE_INDEX data provided in update messages

Name	Value
TBLOWNER	String value of the owner of the table on which the index was created.
TBLNAME	String value of the name of the table on which the index was created.
IXNAME	String value of the name of the created index.
INDEX_TYPE	String value representing the index type: "P" (primary key), "F" (foreign key), or "R" (regular).
INDEX_METHOD	String value representing the index method: "H" (hash), "T" (range), or "B" (bit map).
UNIQUE	Boolean value indicating whether the index is unique.
HASH_PAGES	Integer value representing the number of pages in a hash index (not specified for range indexes).
COLUMNS	String value describing the columns in the index. Format: <col1name>[;<col2name> [;<col3name>[;...]]]

DROP_INDEX

Messages with __TYPE=4 (XlaConstants.DROP_INDEX) indicate that an index has been dropped. Table 6–6 shows the name and value pairs that are included in a MapMessage generated for a DROP_INDEX operation.

Table 6–6 DROP_INDEX data provided In update messages

Name	Value
OWNER	String value of the owner of the table on which the index was dropped.
TABLE_NAME	String value of the name of the table on which the index was dropped.
INDEX_NAME	String value of the name of the dropped index.

ADD_COLUMNS

Messages with __TYPE=5 (XlaConstants.ADD_COLUMNS) indicate that a table has been altered by adding new columns. Table 6–7 shows the name and value pairs that are included in a MapMessage generated for a ADD_COLUMNS operation.

Table 6–7 ADD_COLUMNS data provided in update messages

Name	Value
OWNER	String value of the owner of the altered table.
NAME	String value of the name of the altered table.
PK_COLUMNS	String value containing the names of the columns in the primary key for this table. If the table has no primary key, the PK_COLUMNS value is not specified. Format: <col1name>[;<col2name> [;<col3name>[;...]]]
COLUMNS	String value containing the names of the columns added to the table. Format: <col1name>[;<col2name> [;<col3name>[;...]]] **Note:** For each added column, additional name and value pairs that describe the column are included in the MapMessage.
*_column_name*_TYPE	Integer value representing the data type of this column. From java.sql.Types.
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC or DECIMAL).
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC or DECIMAL).
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR, VARCHAR, BINARY, or VARBINARY).
*_column_name*_NULLABLE	Boolean value indicating whether this column can have a NULL value.
*_column_name*_OUTOFLINE	Boolean value indicating whether this column is stored in the inline or out-of-line part of the tuple.
*_column_name*_INPRIMARYKEY	Boolean value indicating whether this column is part of the primary key of the table.

DROP_COLUMNS

Messages with __TYPE=6 (XlaConstants.DROP_COLUMNS) indicate that a table has been altered by dropping existing columns. Table 6–8 shows the name and value pairs that are included in a MapMessage generated for a DROP_COLUMNS operation.

Table 6–8 DROP_COLUMNS data provided in update message

Name	Value
OWNER	String value of the owner of the altered table.
NAME	String value of the name of the altered table.

Table 6–8 (Cont.) DROP_COLUMNS data provided in update message

Name	Value
COLUMNS	String value containing the names of the columns dropped from the table. Format: <col1name>[;<col2name> [;<col3name>[;...]]] **Note:** For each dropped column, additional name and value pairs that describe the column are included in the MapMessage.
*_column_name*_TYPE	Integer value representing the data type of this column. From java.sql.Types.
*_column_name*_PRECISION	Integer value containing the precision of this column (for NUMERIC or DECIMAL).
*_column_name*_SCALE	Integer value containing the scale of this column (for NUMERIC or DECIMAL).
*_column_name*_SIZE	Integer value indicating the maximum size of this column (for CHAR, VARCHAR, BINARY, or VARBINARY).
*_column_name*_NULLABLE	Boolean value indicating whether this column can have a NULL value.
*_column_name*_OUTOFLINE	Boolean value indicating whether this column is stored in the inline or out-of-line part of the tuple.
*_column_name*_INPRIMARYKEY	Boolean value indicating whether this column is part of the primary key of the table.

CREATE_VIEW

Messages with __TYPE=14 (XlaConstants.CREATE_VIEW) indicate that a materialized view has been created. Table 6–9 shows the name and value pairs that are included in a MapMessage generated for a CREATE_VIEW operation.

Table 6–9 CREATE_VIEW data provided in update messages

Name	Value
OWNER	String value of the owner of the created view.
NAME	String value of the name of the created view.

DROP_VIEW

Messages with __TYPE=15 (XlaConstants.DROP_VIEW) indicate that a materialized view has been dropped. Table 6–10 shows the name and value pairs that are included in a MapMessage generated for a DROP_VIEW operation.

Table 6–10 DROP_VIEW data provided in update messages

Name	Value
OWNER	String value of the owner of the dropped view.
NAME	String value of the name of the dropped view.

CREATE_SEQ

Messages with __TYPE=16 (XlaConstants.CREATE_SEQ) indicate that a sequence has been created. Table 6–11 shows the name and value pairs that are included in a MapMessage generated for a CREATE_SEQ operation.

Table 6–11 CREATE_SEQ data provided in update messages

Name	Value
OWNER	String value of the owner of the created sequence.
NAME	String value of the name of the created sequence.
CYCLE	Boolean value indicating whether the CYCLE option was specified on the new sequence.
INCREMENT	A long value indicating the INCREMENT BY option specified for the new sequence.
MIN_VALUE	A long value indicating the MINVALUE option specified for the new sequence.
MAX_VALUE	A long value indicating the MAXVALUE option specified for the new sequence.

DROP_SEQ

Messages with __TYPE=17 (XlaConstants.DROP_SEQ) indicate that a sequence has been dropped. Table 6–12 shows the name and value pairs that are included in a MapMessage generated for a DROP_SEQ operation.

Table 6–12 DROP_SEQ data provided in update messages

Name	Value
OWNER	String value of the owner of the dropped table.
NAME	String value of the name of the dropped table.

CREATE_SYNONYM

Messages with __TYPE=19 (XlaConstants.CREATE_SYNONYM) indicate that a synonym has been created. Table 6–13 shows the name and value pairs that are included in a MapMessage generated for a CREATE_SYNONYM operation.

Table 6–13 CREATE_SYNONYM data provided in update messages

Name	Value
OWNER	String value of the owner of the created synonym.
NAME	String value of the name of the created synonym.
OBJECT_OWNER	String value of the schema of the object for which you are creating a synonym.
OBJECT_NAME	String value of the name of the object for which you are creating a synonym.
IS_PUBLIC	Boolean value that is TRUE if the synonym is public, or FALSE if not.
IS_REPLACE	Boolean value that is TRUE if the synonym was created using CREATE OR REPLACE, or FALSE otherwise.

DROP_SYNONYM

Messages with __TYPE=20 (XlaConstants.DROP_SYNONYM) indicate that a synonym has been dropped. Table 6–14 shows the name and value pairs that are included in a MapMessage generated for a DROP_SYNONYM operation.

Table 6–14 DROP_SYNONYM data provided in update messages

Name	Value
OWNER	String value of the owner of the dropped synonym.
NAME	String value of the name of the dropped synonym.
IS_PUBLIC	Boolean value that is TRUE if the synonym was public, or FALSE if not.

TRUNCATE

Messages with __TYPE=18 (XlaConstants.TRUNCATE) indicate that a table has been truncated. All rows in the table have been deleted. Table 6–15 shows the name and value pairs that are included in a MapMessage generated for a TRUNCATE operation.

Table 6–15 TRUNCATE data provided in update messages

Name	Value
OWNER	String value of the owner of the truncated table.
NAME	String value of the name of the truncated table.

Data type support

This section covers data type considerations for JMS/XLA.

Data type mapping

Table 6–16 lists access methods for the data types supported by TimesTen. For more information about data types, see "Data Types" in *Oracle TimesTen In-Memory Database SQL Reference*.

Table 6–16 Data Type Mapping

TimesTen column type	Read with MapMessage method...
CHAR(*n*)	getString()
VARCHAR(*n*)	getString()
NCHAR(*n*)	getString()
NVARCHAR(*n*)	getString()
NVARCHAR2(*n*)	getString()
DOUBLE	getDouble()
FLOAT	getFloat()
DECIMAL(*p*,*s*)	getString() Can be converted to BigDecimal or to Double by the application.

Table 6–16 (Cont.) Data Type Mapping

TimesTen column type	Read with MapMessage method...
NUMERIC(*p*,*s*)	getString() Can be converted to BigDecimal or to Double by the application.
INTEGER	getInt()
SMALLINT	getShort()
TINYINT	getShort()
BIGINT	getLong()
BINARY(*n*)	getBytes()
VARBINARY(*n*)	getBytes()
DATE	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970). Can be converted to Date or Calendar by the application.
TIME	getString() Can be converted to Date or Calendar by the application.
TIMESTAMP	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970). It truncates nanoseconds. Use getString() if you require nanosecond precision. Can be converted to Date or Calendar by the application.
TT_CHAR	getString()
TT_VARCHAR	getString()
TT_NCHAR	getString()
TT_NVARCHAR	getString()
ORA_CHAR	getString()
ORA_VARCHAR2	getString()
ORA_NCHAR	getString()
ORA_NVARCHAR2	getString()
VARCHAR2	getString()
TT_TINYINT	getShort()
TT_SMALLINT	getShort()
TT_INTEGER	getInt()
TT_BIGINT	getLong()
BINARY_FLOAT	getFloat()
BINARY_DOUBLE	getDouble()
REAL	getFloat()
NUMBER	getString()
ORA_NUMBER	getString()
TT_DECIMAL	getString()

Table 6–16 (Cont.) Data Type Mapping

TimesTen column type	Read with MapMessage method...
TT_TIME	getString()
TT_DATE	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
TT_TIMESTAMP	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
ORA_DATE	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970).
ORA_TIMESTAMP	getLong(),getString() The getLong() method returns microseconds since epoch (00:00:00 UTC, January 1, 1970). It truncates nanoseconds. Use getString() if you require nanosecond precision.
TT_BINARY	getBytes()
TT_VARBINARY	getBytes()
ROWID	getBytes(),getString()

Data types character set

JMS/XLA uses a UTF-16 character set for the following data types:

- TT_CHAR
- TT_VARCHAR
- ORA_CHAR
- ORA_VARCHAR2
- TT_NCHAR
- TT_NVARCHAR
- ORA_NCHAR
- ORA_NVARCHAR2
- NCHAR
- NVARCHAR
- NVARCHAR2

JMS classes for event handling

You can use JMS classes when programming to the JMS/XLA API. The JMS/XLA API supports only publish/subscribe messaging. JMS classes include the following:

- Message (parent class only)
- TopicConnectionFactory
- Topic
- TopicSubscriber

- Connection
- Session
- ConnectionMetaData
- MapMessage
- TopicConnection
- TopicSession
- ConnectionFactory
- Destination
- MessageConsumer
- ExceptionListener

See the following location for documentation of these classes:

http://download.oracle.com/javaee/5/api/

JMS/XLA replication API

The TimesTen com.timesten.dataserver.jmsxla package includes the TargetDataStore interface and the TargetDataStoreImpl class.

See *Oracle TimesTen In-Memory Database JMS/XLA Java API Reference* for information.

TargetDataStore interface

This interface is used to apply XLA update records from a source database to a target database. The source and target database schema must be identical for the affected tables.

This interface includes the methods shown in Table 6–17.

Table 6–17 TargetDataStore methods

Method	Description
apply()	Applies XLA update descriptor to the target database.
close()	Closes the connections to the database and releases the resources.
commit()	Performs a manual commit.
getAutoCommitFlag()	Returns the value of the autocommit flag.
getConnectString()	Returns the database connection string.
getUpdateConflictCheckFlag()	Returns the value of the flag for checking update conflicts.
isClosed()	Checks whether the object is closed.
isDataStoreValid()	Checks whether the database is valid.
rollback()	Rolls back the last transaction.
setAutoCommitFlag()	Sets the flag for autocommit during apply.
setUpdateConflictCheckFlag()	Sets the flag for checking update conflicts during apply.

TargetDataStoreImpl class

This class creates connections and XLA handles for a target database. It implements the `TargetDataStore` interface.

JMS message header fields

Table 6–18 shows the JMS message header fields provided by JMS/XLA.

Table 6–18 JMS/XLA header fields

Header	Contents
`JMSMessageId`	The transaction log file number of the XLA record.
`JMSType`	The string representation of the `__TYPE` field.

Index

A

B

C

D

E

F

G

H

I

J

L

M

O

P

Q

R

S

T

U

W

X